

Visualization, Adaptation, and Transformation of Procedural Grammars

THÈSE N° 7627 (2017)

PRÉSENTÉE LE 24 MARS 2017

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE D'INFORMATIQUE GRAPHIQUE ET GÉOMÉTRIQUE
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Stefan LIENHARD

acceptée sur proposition du jury:

Dr R. Bouluc, président du jury
Prof. M. Pauly, directeur de thèse
Prof. B. Neubert, rapporteur
Prof. M. Wimmer, rapporteur
Prof. W. Jakob, rapporteur



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2017

In memoriam
Ulrich Lienhard
1948 – 2013

Abstract

Procedural shape grammars are powerful tools for the automatic generation of highly detailed 3D content from a set of descriptive rules. It is easy to encode variations in stochastic and parametric grammars, and an uncountable number of models can be generated quickly. While shape grammars offer these advantages over manual 3D modeling, they also suffer from certain drawbacks. We present three novel methods that address some of the limitations of shape grammars. First, it is often difficult to grasp the diversity of models defined by a given grammar. We propose a pipeline to automatically generate, cluster, and select a set of representative preview images for a grammar. The system is based on a new view attribute descriptor that measures how suitable an image is in representing a model and that enables the comparison of different models derived from the same grammar. Second, the default distribution of models in a stochastic grammar is often undesirable. We introduce a framework that allows users to design a new probability distribution for a grammar without editing the rules. Gaussian process regression interpolates user preferences from a set of scored models over an entire shape space. A symbol split operation enables the adaptation of the grammar to generate models according to the learned distribution. Third, it is hard to combine elements of two grammars to emerge new designs. We present design transformations and grammar co-derivation to create new designs from existing ones. Algorithms for fine-grained rule merging can generate a large space of design variations and can be used to create animated transformation sequences between different procedural designs. Our contributions to visualize, adapt, and transform grammars makes the procedural modeling methodology more accessible to non-programmers.

Keywords: rule-based procedural modeling, shape grammar, best view, thumbnail gallery, view attribute, pdf design, design transformation

Zusammenfassung

Prozedurale Figur-Grammatiken (Shape Grammars) sind hilfreich, um automatisch hochdetaillierte 3D Inhalte zu generieren. Mit stochastischen und parametrischen Grammatiken kann man einfach Variationen beschreiben und schnell viele Modelle erzeugen. Zwar haben Figur-Grammatiken diese Vorteile gegenüber manueller 3D-Modellierung, aber sie haben auch etliche Nachteile. Wir stellen drei neue Techniken vor, die einige dieser Limitationen beheben. Erstens ist es oft schwierig, die Vielfalt der Modelle zu verstehen, die in einer Grammatik definiert sind. Wir präsentieren eine Pipeline, die für eine Grammatik automatisch eine Reihe repräsentativer Vorschaubilder erzeugt, gruppiert und auswählt. Das System basiert auf einem neuen Deskriptor für Ansichtsattribute, der misst, wie gut eine gewisse Ansicht ein Modell repräsentiert und der verschiedene Modelle der gleichen Grammatik vergleichen kann. Zweitens entspricht die Wahrscheinlichkeitsverteilung der Modelle einer stochastischen Grammatik häufig nicht den Wünschen des Benutzers. Ein neues System von uns erlaubt es, neue Verteilungen für Grammatiken zu gestalten ohne deren Regeln bearbeiten zu müssen. Der Benutzer bewertet eine Reihe von Modellen und mittels Gauss-Prozess-Regression wird diese Benutzerpräferenz über den ganzen Figurenraum interpoliert. Mit einer Symbol-Aufspaltungsoperation wird die Grammatik so angepasst, dass neue Modelle gemäss der gelernten Verteilung generiert werden. Drittens ist es nicht einfach, Elemente von zwei Grammatiken zu kombinieren, um neue Designs zu kreieren. Mit Designtransformationen und Grammatik-Koderivation können bestehende Designs zu neuen Designs vermischt werden. Wir präsentieren Algorithmen, um Produktionsregeln stufenlos zu mischen. Damit kann man grosse Räume von Designvariationen aufspannen und animierte Transformationssequenzen zwischen prozeduralen Modellen erzeugen. Mit unseren Kontributionen zur Visualisierung, Adaption und Transformation von Grammatiken können wir Nicht-Programmierern prozedurale Modellierungstechniken näherbringen.

Stichwörter: regelbasierte prozedurale Modellierung, Figur-Grammatik, beste Ansicht, Thumbnailgalerie, Ansichtsattribut, Verteilungs-Gestaltung, Designtransformation

Acknowledgements

I would like to express my deepest gratitude, in no particular order, to the following individuals for their tireless technical, scientific, and moral support: Adi, Alex, Alexandre, Alina, Anastasia, Andi, Ändu, Andrea, Annemarie, Arash, Beni, Bensch, Boris, Cheryl, Chizuko, Chris, Christian, Christiane, Christos, D. Gerb, Dani, Davide, Dävu, Dec, Derek, Duygu, Eliot, Ella, Emily, Erik, Erika, Eva, Fäbu, Frieder, Gaspard, Heikki, Helen, James, Jin, Kusi, Ladi, Laura, Luci, Madeleine, Mano, Mario, Marc, Mark, Mark, Mark, Markus, Martin, Matt, Melissa, Michael, Mina, Minh, Neda, Nik, Nils, Niranjana, Noémie, Oli, Pädä, Pascal, Peter, Ralph, Régis, Sandra, Sebastian, Sharon, Sherpa, Sofien, Steff, Stephan, Tain, Tom, Ueli, Umlaut, Ursi, Vicky, Wädi, Ward, Yannick, Zeno, and Zhou.

Lausanne, January 8 2017

Stefan Lienhard

Table of Contents

Abstract (English/Deutsch)	v
Acknowledgements	ix
Table of Contents	xi
1 Introduction	15
1.1 Contributions	18
1.2 Overview	19
2 A Short History of Grammar-based Procedural Modeling	21
2.1 Lindenmayer Systems	22
2.2 Classical Shape Grammars	23
2.3 Modern Set Grammars	24
2.4 Grammar-based Inverse Procedural Modeling	27
3 Thumbnail Galleries for Procedural Models	31
3.1 Introduction	31
3.2 Related Work	34
3.3 View Attributes	36
3.3.1 Geometric View Attributes	37
3.3.2 Aesthetic View Attributes	38
3.3.3 Semantic View Attributes	40
3.4 Thumbnail Gallery Generation	41
3.4.1 Best View Selection	41
3.4.2 Stochastic Sampling of Rule Parameters	43
3.4.3 Clustering View Attributes	46
3.4.4 Thumbnail Gallery Creation	46
3.5 Results	47
3.5.1 Best View	47

Table of Contents

3.5.2	Clustering	49
3.6	Discussion and Future Work	49
3.7	Conclusions	51
4	Designing Probability Density Functions for Shape Grammars	53
4.1	Introduction	53
4.2	Related Work on Exploratory Modeling	55
4.3	Overview	57
4.3.1	Framework Overview	57
4.3.2	Grammar Definitions	57
4.4	Learning the Probability Density Function	59
4.4.1	Features	59
4.4.2	Gaussian Process Regression (GPR)	61
4.4.3	Preference Function Factorization	65
4.5	Generating Models According to a PDF	65
4.6	User Interface	69
4.7	Evaluation and Results	70
4.7.1	Urban Planning Use Case	77
4.8	Limitations and Future Work	79
4.9	Conclusions	80
5	Design Transformations for Rule-based Procedural Modeling	83
5.1	Introduction	84
5.2	Related Work	86
5.3	Overview	87
5.3.1	Grammar Definitions	87
5.3.2	Framework Overview	88
5.4	Co-Derivation of Shape Grammars	88
5.4.1	Sparse Correspondences	88
5.4.2	Grammar Co-Derivation	89
5.4.3	Rule Merging	90
5.4.4	Rule Merging for Split Rules	93
5.5	Applications & Results	96
5.5.1	Variety Generation with Multiple Grammars	96
5.5.2	Transformation Sequences	99
5.5.3	Quantitative Results	102
5.6	Discussion	103
5.7	Limitations and Future Work	105

5.8	Conclusions	105
6	Conclusions	107
6.1	Future Work	108
A	Detailed Design Transformation Result Descriptions	111
A.1	Sternwarte Chain Part 1	111
A.2	Tree L-Systems	115
	Bibliography	119
	Curriculum Vitae	133

1 Introduction

The need for 3D content is ubiquitous nowadays. Highly detailed models are required in various domains such as medical applications, urban planning, and the entertainment industry. The demand for more models with more details is ever increasing. This, however, leads to a significant overhead because 3D modeling is a non-trivial and time-consuming task when done manually. *Procedural modeling* helps generate models faster, especially when a large number of models of a certain type is needed. In computer graphics, procedural modeling is an umbrella term for methods that encode the modeling process in algorithmic procedures. This has been used to automate the generation of images, 3D models, textures, and even music. The most common procedurally generated 3D contents are fractals, plants, city layouts, buildings, and terrains. Procedural modeling has also been widely adopted by the industry in such products as *speedtree*¹, *CityEngine*², *Terragen*³, or *Houdini*⁴ just to name a few.

When you need only one model, it might be faster to create it in a classical 3D modeling tool because there is an overhead for authoring a procedural description of a model. However, as the number of required models increases, one quickly reaches a point where it becomes more efficient to use procedural modeling. For example, procedural modeling pays off if one wants to generate a large enough variety of building models to fill a virtual city. A benefit of the procedural description is that it can be parameterized and/or randomized to generate different variations of a model. The procedures can then generate any number of these variations at no extra cost. If one were to model these buildings manually, the workload would increase with each additional building.

¹speedtree: <http://www.speedtree.com>, accessed on 2017/02/28

²Esri CityEngine: <http://www.esri.com/software/cityengine>, accessed on 2017/02/28

³Terragen: <http://planetside.co.uk>, accessed on 2017/02/28

⁴SideFX Houdini: <http://www.sidefx.com/products/houdini-fx>, accessed on 2017/02/28

Yet another advantage of the algorithmic description is that it can be encoded in a very compact way. Gigabytes of geometric data for a 3D city can be compressed into a few kilobytes for storing the procedures. Good examples are video games like *No Man's Sky*⁵ in which the player can explore an (almost) infinite universe of different unique planets or *kkrieger*⁶ that stores a complete first-person shooter in just 96 kB.

Most procedural modeling techniques use sets of rules that describe how a coarse model can be replaced gradually with more detailed components. Recursive application of rules leads to more and more details. Rules are often defined with formal grammars, hence terms like *grammar-based* or *shape grammar* are commonly used to name such procedural modeling techniques. Chap. 2 presents a summary of these rule-based techniques that this dissertation focuses on.

While rule-based and procedural techniques in general have many advantages, there are also certain drawbacks. Some programming knowledge is necessary to write procedures or rules to describe content. Traditional artists are often unfamiliar with programming while many programmers lack the artistic sense. Further, it is hard to gain an understanding of a procedural model when only given the rules and not their output. Especially for procedural models with randomized components, i.e., with so-called stochastic rules that offer several different solutions for the same task, it can be hard to visualize and control the space of possible designs they span. These problems are hard in general and not only for people without programming knowledge. Even experienced procedural modelers will have difficulty with certain design goals because it is often not obvious how to encode a procedural model.

In this dissertation we propose three novel algorithmic tools that overcome some of the limitations of procedural modeling and that make shape grammars more accessible to non-programmers. The following paragraphs give an overview of these three methods.

Visualizing Procedural Models It is hard to image what kind of models a shape grammar can produce when only reading the rules. Once the grammar has several parameters or stochastic rules, it becomes even harder to understand what the set of possible models looks like. It could be a very narrow set of similar models, e.g., buildings of a specific architectural style, or it could be a wide set that spans a range of entirely different types of objects.

⁵No Man's Sky: www.no-mans-sky.com, accessed on 2017/02/28

⁶kkrieger: <http://www.farb-rausch.de/prod.py?which=114>, accessed on 2017/02/28

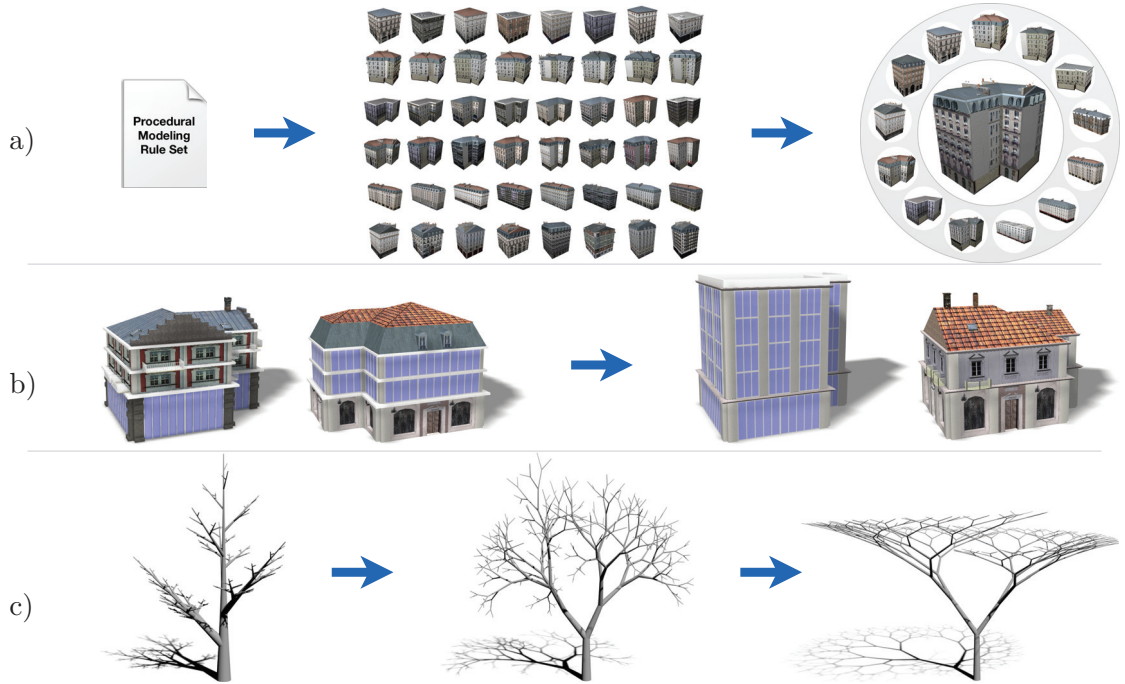


Figure 1.1: Teaser An overview of our three main contributions: a) A pipeline that automatically generates, groups, and selects a set of representative thumbnail images for a grammar. b) A generic building grammar can create houses with mismatching architectural elements. Our framework adapts the distribution of the models generated by the grammar to avoid such undesired models. c) Design transformations allow fine-grained merging of procedural models to emerge new designs. This can be used to smoothly transform one design into another.

In Chap. 3 we present a pipeline that generates preview thumbnail galleries for given grammars. It samples the space of possible models of the grammar, picks representative models, and shows them from their best viewpoint. An example is shown in Fig. 1.1 a). The pipeline is based on a descriptor developed to compare models and to also find their best viewpoint. This is useful for grammar databases. Browsing and exploring such a database is facilitated when each entry is annotated with a thumbnail gallery that shows representative previews.

Adapting Stochastic Grammars With a grammar that has stochastic rules it is possible that models get generated that are undesirable for a certain design goal. For example, with a generic building grammar one might get a glass house with a mismatching old red tile roof or a building that mixes an old style ground floor with modern style upper floors.

Also with stochastic grammars, some designs might be more likely than others. The probability distribution of a given grammar might be undesired. In Chap. 4 we propose a method that allows changing the probability distribution of the models generated by a grammar. By scoring sample models of the grammar, we can tell the system how desired certain designs are. Using machine learning, our system learns a new probability distribution and adapts the given grammar to the desired preferences.

For example, we might want to subdivide a city into distinct districts such financial, residential, or industrial. Given a generic grammar that randomly combines elements from several architectural styles, our system can learn distinct grammars, each one adapted to its district (see Fig. 1.1 b)).

Transforming Rule-based Models Given two grammars, it is difficult to combine elements of both to emerge new designs. A deep understanding of the grammar code is necessary to merge different components into a new grammar, and this requires time. Chapter Chap. 5 introduces design transformations that allow non-programmers to generate a large variety of models from a small set of example grammars. Our system achieves that by merging the outputs of the grammar rules at different levels of the derivation process.

Additionally, with fine-grained control over the merging process, we have the possibility to generate smooth animation sequences between different procedural designs such as in Fig. 1.1 c).

1.1 Contributions

This dissertation is based on and uses parts of the following three publications written in the course of my PhD:

1. LIENHARD S., SPECHT M., NEUBERT B., PAULY M., MÜLLER P.:
Thumbnail Galleries for Procedural Models.
Computer Graphics Forum (Eurographics) (2014). [LSN*14]
2. DANG M., LIENHARD S., CEYLAN D., NEUBERT B., WONKA P., PAULY M.:
Interactive Design of Probability Density Functions for Shape Grammars.
ACM Trans. Graph. (Siggraph Asia) (2015). [DLC*15]

3. LIENHARD S., LAU C., MÜLLER P., WONKA P., PAULY M.:
Design Transformations for Rule-based Procedural Modeling.
Computer Graphics Forum (Eurographics) (2017). [LLM*17]

In summary, the contributions of this dissertation are:

- A descriptor for procedural models that allows finding the best viewpoint and comparing models, which is used in a pipeline that clusters a grammar’s shape space and that picks a set of representative preview thumbnails.
- A system that uses Gaussian process regression to learn a probability distribution for a grammar from preference scores and that adapts the grammar to generate models according to that probability distribution. My contribution focuses on the design of the features used to compare two procedural models and on the adaptation of a grammar to a target probability distribution.
- Algorithms for fine-grained rule merging that can generate a large space of design variations and that can be used to create animated transformation sequences between different procedural designs.

1.2 Overview

The remainder of this dissertation first provides a compact summary of rule-based procedural modeling techniques (Chap. 2) before three main chapters explain how to visualize grammars with thumbnail galleries (Chap. 3), how to adapt the probability distribution of grammars to user preferences (Chap. 4), and how to enable gradual transformations between two procedural designs (Chap. 5). More detailed descriptions for some of the transformation results are provided in App. A. The dissertation is wrapped up in a concise summary and an outlook into the future (Chap. 6).

2 A Short History of Grammar-based Procedural Modeling

Grammar-based or rule-based procedural modeling techniques have their roots in language theory. Noam Chomsky pioneered that field and introduced *formal grammars* [Cho56]. Such a formal grammar G is a string rewriting system defined by a quadruple $\langle NT, T, \omega, P \rangle$ that consists of the set of *non-terminal* symbols NT , the set of *terminal* symbols T , the so-called *axiom* or *start symbol* $\omega \in NT$, and the set of *production rules* P . Each production rule is of the form:

$$(NT \cup T)^* NT (NT \cup T)^* \rightarrow (NT \cup T)^*.$$

The left-hand side or *predecessor* maps a string of symbols to a *successor* string on the right-hand side. Starting from the axiom, strings can be derived by subsequent application of production rules. A string that contains only terminal symbols is called a *sentence* and cannot be derived any further. The *language* $\mathcal{L}(G)$ is defined as all sentences that are reachable from the axiom.

Chomsky also proposed the *Chomsky hierarchy* [Cho56] to categorize different subsets of formal grammars (Tab. 2.1). The types differ by constraints that they enforce on the

Type	Name	Production Rule Form
type 3	(left) regular	$A \rightarrow a B$ and $A \rightarrow a$ and $A \rightarrow \epsilon$
type 2	context-free	$A \rightarrow \gamma$
type 1	context-sensitive	$\alpha A \beta \rightarrow \alpha \gamma \beta$
type 0	recursively enumerable	$\alpha \rightarrow \beta$

Table 2.1: Chomsky Hierarchy Classification of different types of formal grammars. $A, B \in NT$; $a \in T$; ϵ is the empty string; and $\alpha, \beta, \gamma \in (NT \cup T)^*$.

production rules. The lower the type, the more powerful and expressive the grammars are. Each type x is a superset of all types higher than x . Type 0 includes all formal grammars. Most commonly, procedural modeling in computer graphics uses context-free (type 2) grammars. Less often, context-sensitive (type 1) grammars are used. The production rules of these types always replace a single non-terminal symbol with a new string. The derivation of a grammar creates a *parse tree* or *derivation tree*. The axiom is at the root and the leaves represent the current string. Whenever a production rule is applied, the successor symbols are added to the derivation tree as the predecessor's children.

Many state-of-the-art procedural modeling systems have extended beyond using formal grammars, but they still rely on production rules. They are strictly speaking not grammar-based anymore, but the computer graphics community still refers to them as such. Their advantages outweigh the fact that they violate the formal definition. We summarize these techniques in the following.

2.1 Lindenmayer Systems

In 1968 the biologist Aristid Lindenmayer introduced *L-systems* [Lin68], a string rewriting technique motivated by plant growth. Unlike formal grammars that apply production rules sequentially, L-systems use a parallel replacement strategy that simultaneously applies production rules to all non-terminals in a string. This idea reflects biological growth where cells also may divide at the same time. Another difference is that most L-systems can derive a string infinitely many times without ever reaching a sentence. However, each intermediate string in the sequence is a valid stage of the growth.

It was only in 1986 that L-systems became popular in computer graphics when Przemysław Prusinkiewicz used *turtle graphics* to produce visual interpretations of the symbol strings [Pru86]. The turtle linearly steps through a string and executes instructions for each symbol. For example, F advances the turtle forward while drawing a line, $+$ and $-$ rotate the turtle to its left or right, $[$ pushes the turtle state onto a stack while $]$ pops it from it, etc. With this methodology one can also draw fractals, and L-systems also exist in context-sensitive form. The book *The Algorithmic Beauty of Plants* [PL90] covers all L-system techniques in detail.

To create variations, *stochastic* or *probabilistic* L-systems define several production rules for the same non-terminal symbol. Such stochastic production rules are annotated with the probability of being selected. All the rule probabilities for a given non-terminal

symbol have to sum up to 1 as shown in the example below.

$$\begin{aligned} F &\xrightarrow{0.4} F[-F]F[+F]F \\ F &\xrightarrow{0.3} F[-F] \\ F &\xrightarrow{0.3} F[+F] \end{aligned}$$

Another enhancement are *parametric* production rules that allow passing a list of parameters to symbols. They are of the form:

$$\text{predecessor}(\text{parameter}_1, \dots, \text{parameter}_n) : \text{condition} \rightarrow \text{sucessor},$$

where *condition* is an optional boolean expression. The production rule can only be applied if the condition is met. The following is an example of a parametric production rule.

$$F(t) : t > 0 \rightarrow F(t-1)[-F(t-1)]$$

It is also common to define global grammar parameters (sometimes called *attributes*). In Chap. 3 we use production rules that are parametric and stochastic. We rely exclusively on stochastic grammars in Chap. 4, while the focus in Chap. 5 is on *deterministic* grammars that can only generate unique derivations.

L-systems have been successfully extended to interact with their virtual environments, e.g., *open L-systems* [PJM94, MP96] can query the availability of light and water during leaf and root development, and they can avoid self collisions. The same mechanism was used to generate road networks that are sensitive to geographic parameters and to enforce geometric constraints [PM01]. Other extensions ease the progression from silhouettes to details and use positional information such as turtle posture [PMKL01]. On the software side, *cpfg* [PHM00, PKMH00] used to be the de facto standard, but more recently Python-based *L-Py* [BPC*12] has appeared.

2.2 Classical Shape Grammars

Another milestone in procedural modeling is *shape grammars*, invented by George Stiny [SG71, Sti75, Sti80]. Most notable is the Palladian grammar [SM78] that designs ground plans for villas. Instead of operating on strings of symbols, shape grammars act on arrangements of graphical primitives like lines, circles, or rectangles. Also production rules have only a geometric representation that maps one set of shapes into another.

Shape grammars of this form are typically derived with a user in the loop since they are not well suited for algorithmic evaluation; there are often an infinite number of ways in which a given production rule can be applied. This is because shapes are not atomic, they can be decomposed and reassembled into new shapes. For example, if the predecessor is a simple line, it can be scaled arbitrarily and positioned anywhere on any line in the current shape to match a subline of it. This highly ambiguous subshape matching problem is in fact NP-hard [YKG*09]. Existing computational methods resort to very simple shapes, do not consider subshapes, or use graphs to represent shapes and subgraph matching to identify the production rules, e.g., Grape [GE11].

Terry Knight presented an interesting extension that transforms shape grammars by defining higher level transformation rules [Kni94]. These do not operate directly on the current shape, but they are applied to the grammar’s production rules. Our work in Chap. 5 is inspired by Knight’s book.

2.3 Modern Set Grammars

A simplification of shape grammars is *set grammars* [Sti82] where production rules act on a set of labeled shapes rather than on graphical primitives. A labeled shape is an atomic element of a set grammar which cannot be disassembled for rule matching. This allows for writing production rules in text form with the labels as symbols. Grammars formulated that way can handle complex 3D geometry. Procedural modeling literature unfortunately never made a distinction between the terms set grammar and shape grammar. The latter is used almost exclusively. We refer to the language of a modern shape set grammar, i.e., all the models it can generate, as *shape space*. The derivation tree of such a grammar is called *shape tree*.

CGA Shape Since set grammars avoid the computational limitations of shape grammars, they gained popularity in computer graphics, mostly for architectural modeling. This is also due to the invention of *split grammars* [WWSR03]. A *split rule* subdivides a shape along an axis into a set of smaller shapes that are tightly aligned. This is especially useful for dividing façades into smaller elements such as floors and window tiles. The most well known split grammar derivative is *CGA Shape* (standing for *Computer Generated Architecture*) [MWH*06] by Müller and colleagues that eventually evolved into the commercial software CityEngine. In CGA a shape has a list of geometric and

non-geometric attributes. The most important ones are encoded by the *scope*, i.e., a local coordinate frame and the shape’s size. Successors of CGA production rules consist of terminal or non-terminal symbols, or of *operations* that modify shapes and their attributes. Besides common operations for affine transformations, CGA further supports roof generation, geometry splitting, occlusion testing, and instancing of asset meshes. Unlike L-systems, CGA derives sequentially in breadth-first order, and it executes the operations during the application of a production rule and not only in a post-process. This is necessary for split rules and occlusion queries because they can only be evaluated if the shape’s current state is known. The common modeling strategy first creates a crude *mass model* that is refined by subsequent rule applications that add more and more details. CGA/CityEngine has been used for digital cultural heritage [HMGV09], and all our results in Chaps. 3 and 4 were created with it.

Derivatives and Extensions CGA-like grammars have been used to construct masonry buildings and to optimize their stability by means of static analysis [WOD09]. Another grammar language is the *Generalized Grammar* (G^2) proposed by Krecklau et al. [KPK10]. With G^2 it is possible to pass non-terminal symbols as parameters to productions rules. This enables more generic rules, e.g., the non-terminal symbol for a window can be passed to and used by a generic façade production rule. G^2 can also transform shapes with free-form deformations (FFD) [Sed86], an idea that was later on improved by Zmugg et al. [ZTK*13, ZTK*14]. An extension to G^2 facilitates the construction of interconnected structures by defining attachment points and linking them together [KK11]. Thaller et al. suggest to generalize scopes to arbitrary convex polyhedra [TKZ*13]. More recent advances lead to *CGA++* [SM15], an enriched version of the CGA syntax that allows querying the shape tree during runtime, adds elements of functional programming, and provides synchronization primitives for more control over the derivation. *Group grammars* [SP16] enable rules that operate on several shapes at once. Arrangements of shapes can be relabeled into distinct patterns, and this is used to create tangle drawings.

Worth mentioning is also the *Generative Modeling Language* (GML) [Hav05]. Even though it is not a proper grammar language, it follows the same methodology of refining a model step-by-step by replacing parts of the model with more detailed subparts. GML uses Euler operations [Bau72] to modify meshes, and is based on PostScript syntax which makes it somewhat cumbersome to use. In general, procedural modeling systems have to make a trade-off between their expressiveness and the complexity of their syntax. Similar in that respect is also a framework by Leblanc et al. [LHP11] that never received

much attention even though it supports some powerful shape editing operations such as constructive solid geometry (CSG).

Interactive Rule-based Modeling Interactive techniques alleviate the authoring and editing process of grammars. Instead of working on the text form, a user can directly interact with the visualization of a grammar. Notable work in that area was done by Lipp et al. [LWW08] with a system that makes it possible to select and modify specific components of a rule-based model or to build one entirely from scratch by just clicking and dragging with the mouse to select rules, to modify split sizes, etc. Follow-up projects by Patow and colleagues also expose the dependencies of the production rules as a directed acyclic graph (DAG) and let the user interact with it [RP12, Pat12, BBP13]. It has been shown that several production rules can be grouped into *high-level primitives* together with handles that let non-programmers adjust and combine them [KK12]. Kelly et al. investigated the best placement of such interaction handles and dimensioning lines for parametric grammars [KWM15]. A visual programming language (VPL) that generates CGA grammars by building a graph of nodes that encapsulate basic CGA operations exists too [SMBC13]. *Guided procedural modeling* [BŠMM11] embeds stochastic L-systems in *guides* that communicate with each other, and the user can deform them.

GPU Grammars It is possible to parallelize the derivation process of grammars, allowing the generation of entire cities or forests in a fraction of a second on massively parallel *graphics processing units* (GPU). Lipp et al. provide a lock free parallel implementation of L-systems [LWW10]. There are several projects that encode façade split grammars in the pixel shader and evaluate it separately for every fragment on the GPU. Haegler et al. [HWMA*10] proofed the concept and showed that inserts and extrusions can be ray-marched in screen space. This was improved by Marvie et al. [MGHS11] with exact intersection calculations, and they used coverage polygons to render extrusions that reach over the façade polygon (in screen space). Alternatively, geometry instantiation can be used for extrusions [KBK13]. The first attempt at deriving entire buildings and cities on the fly on the graphics card was presented by Marvie et al. [MBG*12]. Thanks to the dynamic GPU task scheduling algorithm *Softshell* by Steinberger et al. [SKK*12], *PGA* (*parallel generation of architecture*) [SKK*14a, SKK*14b] can derive and render full cities at much higher performance.

Graph Grammars *Graph grammars* (originally called *web grammars*) [PR69] introduce a topological component. Instead of replacing non-terminal symbols, graph grammars replace nodes or subgraphs of a graph. Similar in concept, but independently developed, are *Plex languages* [Fed71]. They also connect and replace subgraphs at predefined attachments points. Use cases are chemical molecules, logic diagrams, electrical circuits, flowcharts, etc.

2.4 Grammar-based Inverse Procedural Modeling

Inverse procedural modeling is the process of finding a grammar that represents a given 3D model or image. It is a very difficult problem: there are often infinitely many ways in which a certain model can be encoded by a grammar, and it is unclear which one to pick. Applications of inverse procedural modeling are compression, i.e., finding a compact grammar instead of explicitly storing a model, reconstruction from real world data, or generation of variations by changing the parameters of the resulting grammar.

Known inverse techniques can broadly be grouped into three categories, whereof only the methods in the first one really try to solve the full inverse problem and create a new grammar from scratch. The other two categories either fit a generic template grammar to the input data or they infer a stochastic grammar from several input exemplars using Bayesian induction, but that requires structured and annotated input data. Generally, the methods require strong prior knowledge of the input data to work well. For example, many projects infer building façade grammars from images because their underlying regular split structures facilitate the task. In the following, we describe the three categories.

From Scratch There are only few methods that generate new grammars from scratch. The existing ones detect symmetries and similarities in the input model. Subparts that are related by similarity transformations can be grouped together and described by recursive production rules. This has been done for 2D vector drawings consisting of basic primitive shapes such as lines, curves, triangles, etc. [ŠBM*10]. For translations and mirror symmetries only, Bokeloh et al. [BWS10] offer a solution that does the same in 3D using a voxel grid to find similarities.

As mentioned before, the same task for regular façades is often easier since they can be described with a few split rules. Early approaches rely on the user to manually subdivide and label images into semantic elements such as floors and windows that are then encoded

with split rules [BA05, ARB07]. Newer methods do this automatically [MZWG07, WYD*14].

Template Fitting In the second category are methods that try to fit a generic, parametric grammar to some input data. This has been done in various forms for nicely structured façades with a generic split grammar [TSK*10, TKS*11, STKP11, RKT*12]. In 3D, the same approach works for Doric temples [MMWG11]. While such methods work for very specific types of input data, they are very difficult to generalize, e.g., for all buildings. Recently, machine learning techniques have been applied to match sketches to parametric shape grammar snippets that define, e.g., windows, roofs, or building mass models that can be combined to full grammars [NGDA*16].

Related are also methods that work with stochastic grammars and that try to find a specific derivation that optimizes a certain criteria. For example, Talton et al. [TLL*11] use *Markov chain Monte Carlo (MCMC)* to efficiently sample the shape space of an L-system to find the one derivation that is contained within and fills a given 3D volume as much as possible. Stochastic grammars are difficult to control, and volumetric high-level descriptions provide an intuitive way to guide the derivation. Ritchie et al. show an improved sampling technique that speeds up the convergence of the method [RMGH15]. Analogous, although not grammar-based, is the inverse procedural modeling approach by Št’ava et al. [ŠPK*14] that uses a tree distance metric together with MCMC to optimize parameters of a sophisticated botanical tree model in order to generate more similar trees.

Lau et al. [LOMI11] reduce 3D models of furniture to graphs and parse them with simple Plex-like grammars. This way, they can decompose tables and cabinets into their components and tell how to put them together.

Grammar Induction The last category borrows ideas from natural language processing. Given a corpora of example sentences, Bayesian model merging can learn a simple stochastic grammar that can generate the exemplars and also new sentences with similar structure [SO94, Sto94]. In computer graphics, Talton et al. [TYK*12] were the first to apply this method to 3D models. First, trivial deterministic grammars are extracted from a set of annotated scene graphs. Then, compatible production rules can be merged together to make the grammar more general, or stochastic production rules can also be split up again, resulting in a more specific grammar. These two operations in conjunction

with MCMC allow sampling the space of possible grammars. A minimum description length prior imposed on the grammar structure helps to find an optimal grammar that trades off grammar compactness and the probability of generating the input examples. In Chap. 4 we use a rule splitting operation similar to Talton’s to change probability distributions of stochastic grammars. Bayesian grammar induction has of course also been applied to façades [WRPG13, MG13].

3 Thumbnail Galleries for Procedural Models

Procedural modeling allows for the generation of innumerable variations of models from parameterized, conditional or stochastic grammars. Due to the abstractness, complexity and stochastic nature of grammars, it is often very difficult to have an understanding of the diversity of the models that a given grammar defines. We address this problem by presenting a novel system to automatically generate, cluster, rank, and select a series of representative thumbnail images out of a grammar. We introduce a set of *view attributes* that can be used to measure the suitability of an image to represent a model, and allow for comparison of different models derived from the same grammar. To find the best thumbnails, we exploit these view attributes on images of models obtained by stochastically sampling the parameter space of the grammar. The resulting thumbnail gallery gives a representative visual impression of the procedural modeling potential of the grammar. Performance is discussed by means of a number of distinct examples and compared to state-of-the-art approaches.

3.1 Introduction

Rendering and display capabilities are leading to an increasing demand for high quality 3D models. Manually creating a large variety of detailed models is very tedious. Procedural modeling methods help to reduce the manual effort required to define a model, while at the same time providing an efficient way to describe and store a model. Moreover, once a procedural description (i.e., a rule set or a grammar) of a model is obtained, one can easily generate variations of the model by just manipulating a few rule parameters. The diversity of models that can be represented procedurally is large and ranges from plants and furniture, to buildings and up to whole city layouts.

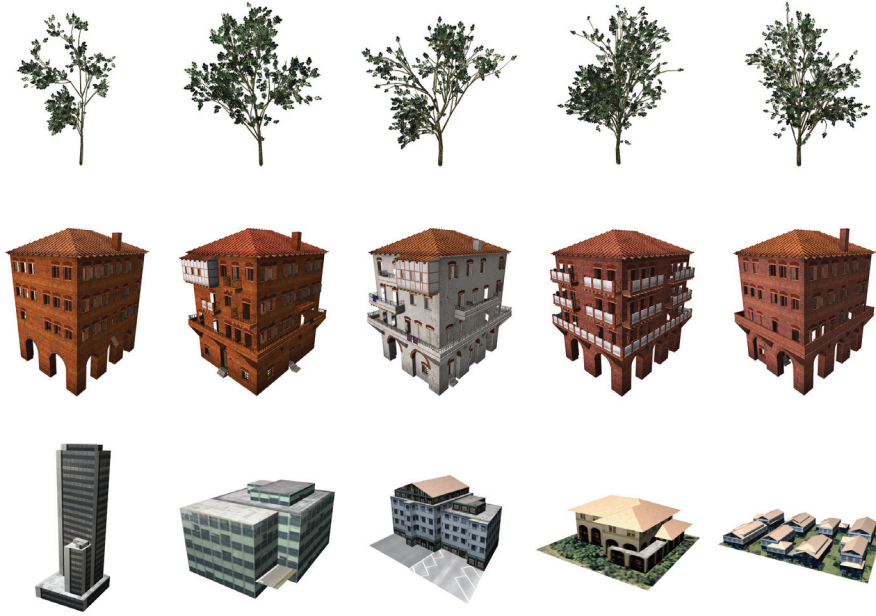


Figure 3.1: *Diversity of Grammars* Each row shows five potential outcomes of one single grammar. A grammar can be stochastic (often with unpredictable design emergence as shown in the top row) or parametric (often with numerous, hard-to-use parameter sets) and produces geometry of arbitrary detail. A rule author is not restricted to a specific content scope, e.g., a rule set can encode variations within a specific building style (middle row) or variations of different building styles (bottom row). As a consequence, the modeling potential of a grammar is difficult to predict, grasp, and represent.

However, one inherent problem of procedurally defined models is that without any additional information, the induced changes of a single parameter are hard to predict and might even have a global scope. In particular, small variations of one parameter might completely change the appearance of a model, while changes of another parameter might only affect details. Furthermore, because parameters are often not independent and rules can be of stochastic nature and contain conditional decisions, it is hard to oversee the vast variety of different models that can be produced from one grammar. Fig. 3.1 visualizes this by means of three different grammars. For each grammar, five models were generated by varying the grammar’s parameters. Note how the differences in the Roman house in the second row are relatively subtle while the buildings in the bottom row are completely different, both in types and numbers.

In order to overcome this problem, we propose a system that automatically generates and arranges a series of thumbnail images into a gallery as illustrated in Fig. 3.2. Such a thumbnail gallery captures the variety of potential designs to show the expressiveness of a given grammar.

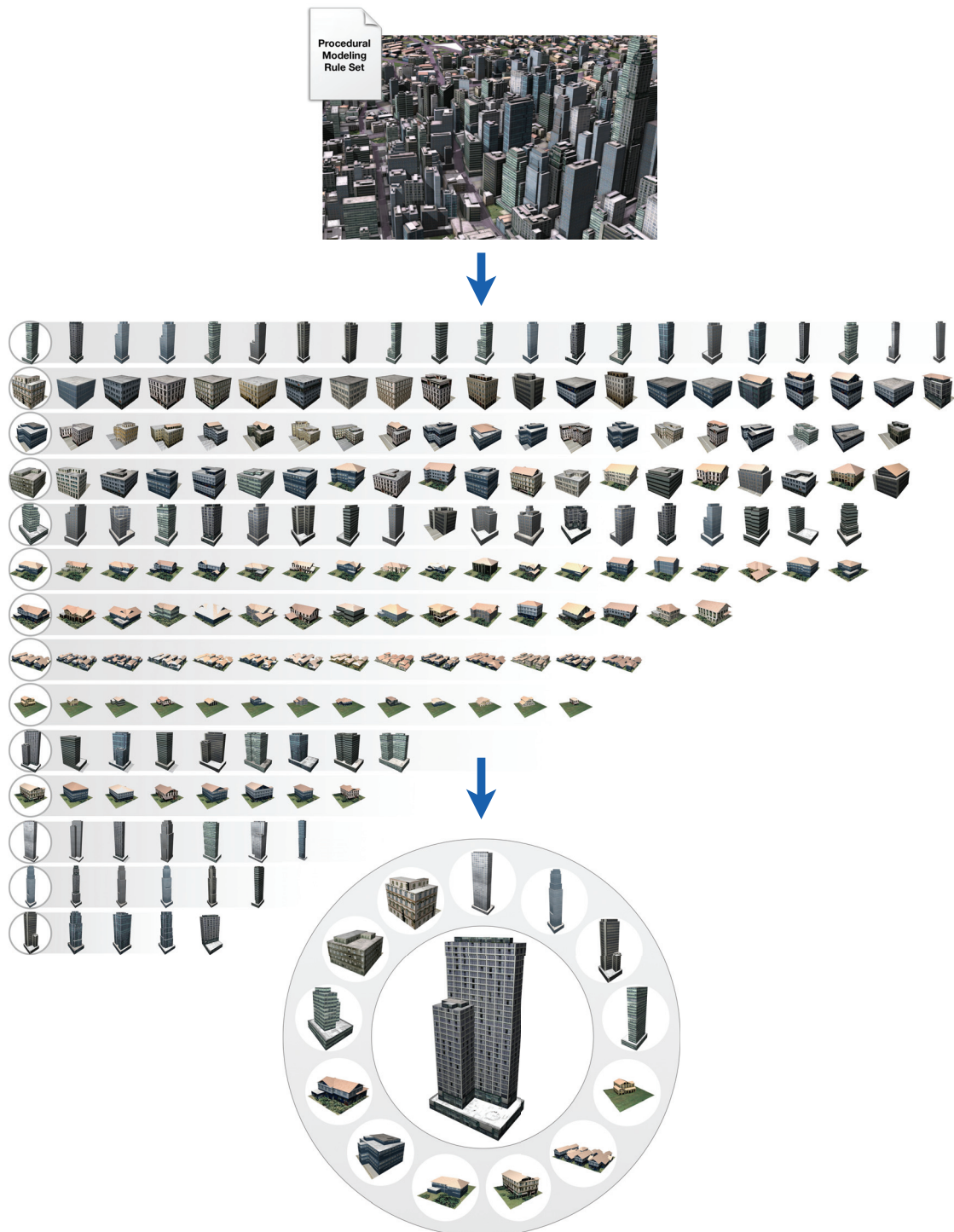


Figure 3.2: Thumbnail Gallery Generation Pipeline In procedural modeling, a grammar or rule set can produce a wide variety of 3D models (top). This chapter presents a thumbnail gallery generation system which automatically samples a grammar, clusters the resulting models into distinct groups (middle), and selects a representative image for each group to visualize the diversity of the grammar (bottom).

To define this set of representative images, we first generate a number of model exemplars by stochastically sampling the grammar’s parameter space. We introduce a set of view attributes that can be used both for finding an optimal view (intra-model variation) and discriminate between different models (inter-model distance). These view attributes can be computed efficiently in image space using programmable graphics hardware. Amongst different views of the same model we chose the most representative view that maximizes a scoring function based on these view attributes.

An adapted weighting scheme of our attributes allows to cluster the representative images of different models. The clusters are ranked and their centers are used to finally depict the model potential of the given grammar in a gallery.

The major contributions of this chapter are:

- A set of normalized view attributes that allow for both best view finding and inter-model comparison.
- A system for automatically generating representative images from a procedural grammar into a thumbnail gallery.

Compared to previous work, our view attributes permit finding better best-views. The gallery generation system exploits the above-mentioned view attributes and opens up the door for a number of more user-friendly procedural modeling applications.

3.2 Related Work

Rule-based procedural modeling was already discussed in detail in Chap. 2. For all examples in this chapter we used CityEngine, but any procedural modeling system could have been applied. We discuss the remaining related work corresponding to the intermediate steps of our system: best view selection, image clustering, and model retrieval.

Best View Selection Some viewpoints reveal and preserve more details of a given model than others. A large body of related work deals with finding the best viewpoint under certain assumptions. Vázquez et al. [VFSH01] define the viewpoint entropy of a given view direction based on the relative area of the model’s projected polygonal faces,

and they successfully apply this quality measure for molecule visualization [VFSL02]. Mesh saliency, introduced by Lee et al. [LVJ05] has the goal to include perception-based metrics in the evaluation of the goodness of a view. A Gaussian-weighted average of mean curvature across different scales is used as a surface feature descriptor and as a measure of local importance, and assigned to each vertex of the mesh. Gooch et al. [GRMS01] introduce a number of heuristics that aim at capturing aesthetic considerations for scene composition. Podolak et al. [PSG*06] select views that minimize symmetry to avoid visual redundancy in the depicted objects. Secord et al. [SLF*11] and Dutagaci et al. [DCG10] evaluate a combination of different view attributes in a user study. While these methods define the quality of a viewpoint with respect to a single model, Laga [Lag10] focuses on finding views that maximally discriminate between different models under the assumption that models belonging to the same class of shapes share the same salient features. Unlike these methods, we exploit view-dependent information to compare different models, and we use the semantic information contained in procedural models to develop new image-based features.

Image Retrieval Clustering images for thumbnail generation requires a metric that quantifies the similarities of different depictions of the grammar. Defining such a metric is a challenging problem, and methods in this area can be grouped based on the features they use to do so. Low level features, such as color [DMK*01, MRF06] or texture and patterns [LP96, HD03] are readily available, while high level features need to be carefully extracted from images (see the survey by Liu et al. [LZLM07] for a detailed overview). In contrast to these methods we are able to employ additional semantic information about our 3D objects given the procedural definition.

3D Model Retrieval Our approach shares the objective to identify similar objects within a group of models with 3D retrieval techniques (see the survey by Bustos et al. [BKS*05] for a detailed overview). A set of features is computed across a database of models that is then used to implement a distance metric and allows for efficient nearest neighbor queries. The features used for this purpose are mostly model driven, and an important goal is to be view independent. In contrast, we actively strive for discriminating features to identify most representative views.

3.3 View Attributes

A *view attribute* is a scalar value that quantifies a certain property of a 3D model seen from a given viewpoint and view direction. Examples are the visible surface, silhouette length, or contrast in the observed image (exact definitions follow). In general we use such view attributes to achieve two different goals:

1. We compare different views of a single model by ranking their view attributes to find the best or most appealing view. To this end, a linear combination of view attributes is used to assign a goodness score to each viewpoint (see Sec. 3.4.1 for details) that we optimize for.
2. As a new contribution, we use the same view attributes to compare and distinguish different models seen from a fixed viewpoint. This inter-model comparison is used to cluster different variations of our procedural models into distinct groups.

Note that our approach is inherently different from classical 3D shape descriptors used for object retrieval. We show that 2D view attributes that work well for best view finding can also be used to compare different derivations of a procedural design seen from the same viewpoint. This inter-model comparison approach is also faster than comparing based on 3D descriptors.

The problem with most existing view attributes is that they are not suited for inter-model comparison. For example, surface visibility, i.e., the ratio of the model’s visible to its total surface [PB96], is a relative value and has no information about the real size of objects. Also the linearity of many existing attributes is also not suited for clustering, it often helps to look at attributes on a logarithmic scale as we will show.

Our scoring function is based on a combination of adapted existing and several new view attributes that are specifically designed to work well for inter-model comparison. In this section we present these view attributes grouped by the different aspects they capture: geometry-based, aesthetic, and semantic view attributes.

To calculate the view attributes for a given viewpoint, the model is rendered only once on the GPU and such information as normals, luminance, and color-coded information about terminals (used for view attributes a_2 , a_7 , and a_8 defined in the following) is stored. We use a technique similar to G-buffers [ST90] inspired by Castello et al. [CSCF06]. All proposed view attributes are computed using solely the stored 2D data and no further analysis of the polygonal geometry is necessary, which allows for very fast evaluation

for a number of different viewpoints. We only look at the geometry in a preprocessing step that stores face and terminal area sizes in lookup tables and the color mapping (color value to face or terminal index) for our color-coded buffers. This information is reused every time we need to compute the view attributes from a new perspective. In our application we want to see the full model as large as possible. To this end, the camera is always placed at the distance where the model’s minimal bounding sphere fits tightly into the frustum. We normalize all view attributes to lie in $[0, 1]$.

3.3.1 Geometric View Attributes

a₁ Pixel Count This view attribute is the ratio of projected area n (in pixels) of the model on the screen to the overall image size [PB96]:

$$a_1 = \frac{4}{\pi} \frac{n}{width^2}.$$

The idea is that the larger the projected area, the more you see of the object. Since our model’s bounding sphere is fit into the view frustum, the projected area can never be larger than a circle with diameter equal to the image side length.

a₂ Surface The ratio of visible to total surface area seen from a specific viewpoint is called surface visibility [PB96]. Maximizing this view attribute minimizes the amount of occluded surface area. For our application, we define the surface view attribute as the logarithm of base $\mathcal{M} = 10^6$ of the visible surface area A in m²:

$$a_2 = \log_{\mathcal{M}}(A + 1).$$

Due to the logarithm there is a higher resolution for lower values of model sizes and a decreasingly lower resolution as the models get bigger. The choice of \mathcal{M} leads to attribute values in $[0, 1]$ (unless we encounter models with visible surface larger than 10⁶ m², e.g., a fictional super skyscraper). Tab. 3.1 shows how a_2 correlates to object size. We increase A by one to be able to assess also small objects.

Example	Max Vis. Surf.	a_2
Furniture	$\approx 1 \text{ m}^2$	0.05
Vehicle	$\approx 10 \text{ m}^2$	0.17
Residential building	$\approx 200 \text{ m}^2$	0.38
Apartment building	$\approx 1000 \text{ m}^2$	0.5
Office building	$\approx 10000 \text{ m}^2$	0.67
High-rise	$\approx 60000 \text{ m}^2$	0.8
City block	$\approx 300000 \text{ m}^2$	0.91

Table 3.1: Surface View Attribute The table lists a_2 for objects of different size seen from one given point.

a₃ Silhouette Length The longer the object’s silhouette is in the rendered image, the more interesting details such as protrusions and concavities should be visible. To bring this value into a reasonable range, we define the view attribute as:

$$a_3 = \log_{16} \left(\frac{s}{l} \right),$$

where s is the silhouette length in pixels, and l is the side length of the largest square that could possibly be rendered (largest square that still fits into the projection of the object’s bounding sphere). We choose 16 as base for the logarithm for the following reason: A standard rectangular object results in an outline of $s \approx 4l$. For this case, we want $a_3 \approx 0.5$ which results in base 16. Furthermore, tests showed that the outline of length $s \approx 16l$ ($a_3 \approx 1$) is an adequate limit for shapes of high complexity with many concavities.

3.3.2 Aesthetic View Attributes

While the aesthetic properties of an image are highly subjective, there exist compositional heuristics, e.g., the widely known *rule of thirds* [Smi97]. Our aesthetic view attributes are based on such artistic guidelines.

a₄ Contrast Higher contrast stands for a larger dynamic range and a visually appealing image. We use root mean square contrast [Pel90] of the rendered image, because it can be computed efficiently:

$$a_4 = \sqrt{\frac{1}{n} \sum_{i=0}^n (I_i - \bar{I})^2},$$

where n is the number of pixels, I_i is the luminance at pixel i , and \bar{I} is the average luminance of all n pixels. We only consider the pixels that have been rasterized and neglect the uniformly colored background.

Since the contrast depends on the lighting, we use the same lighting conditions for all models. We use a standard three point lighting technique, where key, fill, and back light are in fixed positions with respect to the camera and the scene center [Bir00].

a₅ Normal Ratio One artistic composition rule states that the projections of front, side, and top of an object should have relative areas of 4, 2, and 1 in an image [EB92, Arn54]. Gooch et al. [GRMS01] orient the object’s bounding box so that the projections of its three visible sides fulfill that ratio (front and side dimensions can be exchanged). Bounding boxes are only a rough approximation of the true geometry and we analyze image space normals instead. They are grouped into the three categories: left (counting towards the front), right (counting towards the side), and up. While it is possible to compose these three variables into a scalar that quantifies the deviation from the desired ratio, our experience has shown that the same value does not perform well for inter-model comparison. Therefore, we decided to merely distinguish left from right pointing normals and defined the normal ratio as:

$$a'_5 = \frac{l}{l+r}, \quad a_5 = 1 - \left| 1 - \frac{3}{2}a'_5 \right|,$$

where l and r are the number of pixels with normals pointing towards the left or the right respectively. a_5 is used for best view selection and a'_5 for inter-model comparison. The optimal ratio $\frac{l}{r} = \frac{4}{2}$ leads to $a'_5 = \frac{2}{3}$ which maximizes $a_5 = 1$.

a₆ Form Most appealing are renderings for which the object covers a greater part of the image. Degenerate objects, i.e., with very thin renderings are less favorable. The form view attribute is a function of the ratio of the height and the width of the 2D axis aligned bounding box (AABB) of the rendering:

$$a'_6 = \frac{1}{2} \left(\log_l \left(\frac{h}{w} \right) + 1 \right), \quad a_6 = 1 - \left| \log_l \left(\frac{h}{w} \right) \right|,$$

where h and w are the height and the width of the AABB. The base of the logarithm, l , is the side length of the largest possible square (same l as in a_3).

For inter-model comparison we use a'_6 , which allows to differentiate between flat, square, and tall thin AABBs. a_6 is used for best view selection. A square is best while horizontally and vertically thin AABBs are equally unwanted. This view attribute is similar to a_1 for best view finding but it is useful for distinguishing thin tall from wide flat objects.

3.3.3 Semantic View Attributes

The procedurally generated models we are working with are annotated with extra information, e.g., the shape tree or correspondences between geometry and terminal symbols. The following semantic view attributes exploit this extra information.

a₇ Visible Terminal Types Every model is composed of terminal symbols that represent the model’s geometry. Seeing many terminals does not necessarily mean that one obtains a lot of information, because many terminal shapes might be of the same type. What matters most is the number of different terminal symbols that are visible. Most of our examples contain between 2 and 100 terminals (e.g., the Petronas Towers in Fig. 3.6 only uses two terminal types called *glass* and *metal*). We define:

$$a_7 = \log_{100} N_t,$$

where N_t is the number of visible terminal symbols in the image. The logarithm maps that count onto $[0, 1]$ (assuming that there are not more than 100 terminals types). Changes of N_t have more importance for situations with few visible terminal types.

a₈ Terminal Entropy Viewpoint entropy [VFSH01, Váz03] is an adaptation of Shannon entropy, and it is a measurement for the amount of information that one sees from a given viewpoint. It uses as probability distribution the relative area of the projected faces over the sphere: $p_i = \frac{f_i}{A_{\text{tot}}}$, with f_i the projected area of face i and A_{tot} the total projected area of all visible faces (both in solid angles). Our view attribute measures the entropy of visible terminal types. Terminal types provide a more semantic entity of information than polygonal faces:

$$a_8 = - \sum_{i=0}^{N_t} \frac{t_i}{A_{\text{tot}}} \log_{100} \frac{t_i}{A_{\text{tot}}},$$

where t_i is the projected area of terminal type i , and A_{tot} is the total projected area of the object. Both variables are again in solid angle. N_t is again the number of visible terminal types. We use the same base for the logarithm as a_7 , with the same reasoning. Originally, viewpoint entropy used a spherical camera, we use the extension to perspective frustum cameras [VFSL02].

3.4 Thumbnail Gallery Generation

We introduce a system for the automatic creation of a thumbnail gallery for a given grammar. The system is outlined in Alg. 3.1 and consists of the following steps:

1. The default model is procedurally generated in CityEngine (CE) using the default values of the grammar’s parameters. This model is then used to calculate the best viewpoint for the grammar (Sec. 3.4.1).
2. The grammar parameter space is stochastically sampled to generate model variations. Each model is rendered from the previously found viewpoint to obtain its thumbnail image and view attributes (Sec. 3.4.2).
3. The resulting list of view attributes is clustered into distinct groups, and for each group a center is selected. The thumbnail images associated with these centers define the final thumbnail gallery (Sec. 3.4.3).
4. For a comprehensible visualization, the selected representatives are sorted radially around the default model in a reduced 2D space. The latter is obtained by applying a principle component analysis (PCA) on the view attributes (Sec. 3.4.4).

3.4.1 Best View Selection

Once the default model has been created, we search for its best view by sampling a number of viewpoints on a sphere placed around the model. The camera’s view direction always points towards the center of the sphere. We discard viewpoints that lie below the base plane of the model or look down too steeply. We found that good viewpoints were located between 0° and 45° above ground [BTBV96]. The model is rendered from all sample cameras and the view attributes are stored.

Chapter 3. Thumbnail Galleries for Procedural Models

Algorithm 3.1 System Overview

```

# compute best view for default model
default_model = CE.generate(grammar, grammar.default_params)
bestview = computeBestview(default_model, bestview_weights)

# sample grammar's parameter space
⟨thumbnail0, view_attrs0⟩ = render(default_model, bestview)
for  $i = 1$  to  $n\_samples$  do
    params = stochasticSample(grammar.param_ranges)
    model = CE.generate(grammar, params)
    ⟨thumbnail $i$ , view_attrs $i$ ⟩ = render(model, bestview)

# cluster view attributes and select thumbnails
clusters = calcClusters(view_attrs, cluster_weights, n_clusters)
for each cluster  $\in$  clusters do
    if  $0 \notin$  cluster then # skip cluster with default_model
        find  $i$  with view_attrs $i \in$  cluster closest to center of cluster
        add  $i$  to selection

# sort selected thumbnails and create gallery
view_attrs_2D = PCA(view_attrs, 2)
selection.sortRadial(view_attrs_2D $\forall i \in$  selection, view_attrs_2D0)
gallery = thumbnail $0 \cup \forall j \in$  selection

```

To increase stability, the view attributes are normalized to the $[0, 1]$ range, i.e., for a given view attribute, the lowest sampled value will be mapped to 0, the highest value to 1, and in-between values are linearly interpolated. To rank the viewpoints, a score is defined as a linear combination of the normalized view attributes. We empirically determined the weights listed in the second column of Tab. 3.2. More sophisticated schemes for automatically choosing weights, e.g., through semi-supervised learning, are an interesting area of future work [KHS10]. For existing view attributes we started with values that Secord et al. [SLF*11] concluded from a user study.

View Attribute	Best View Weights	Clustering Weights
Pixel count (a_1)	15 %	10 %
Surface (a_2)	20 %	25 %
Silhouette (a_3)	5 %	30 %
Contrast (a_4)	2 %	2 %
Normal ratio (a_5 or a'_5)	10 %	5 %
Form (a_6 or a'_6)	3 %	3 %
Visible terminal types (a_7)	25 %	5 %
Terminal entropy (a_8)	20 %	20 %

Table 3.2: Best View and Clustering Weights The table lists the weights of the view attributes for best view selection and for clustering. Note that normal ratio and form use slightly different view attribute definitions for best view selection and clustering.

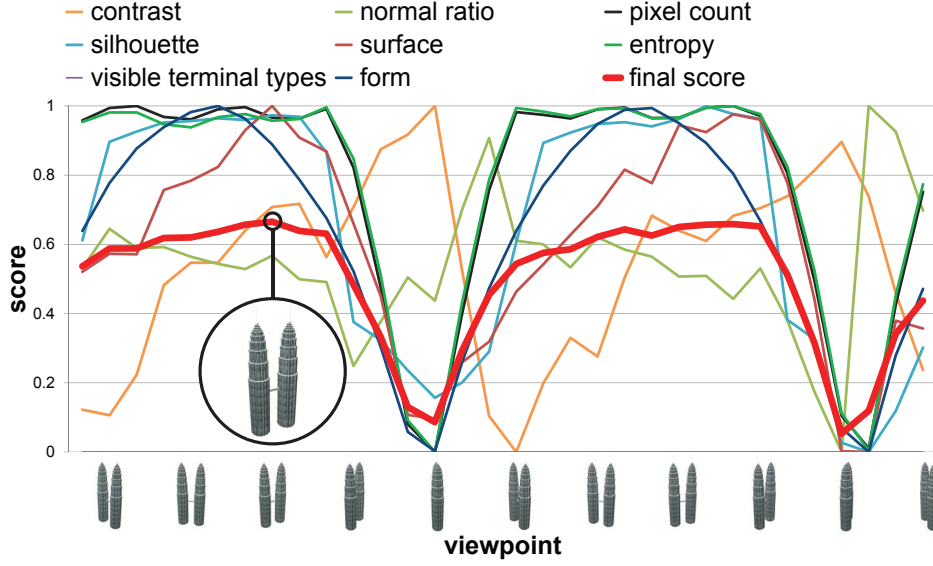


Figure 3.3: View Attribute Graph Normalized view attributes are plotted for different viewpoints for the Petronas Towers grammar. The final score (red) is a linear combination with the weights in Tab. 3.2. Its peak and the corresponding thumbnail are highlighted.

The graph in Fig. 3.3 plots different viewpoints versus the normalized view attributes and the final score. There are 32 viewpoints sampled on a ring $\frac{\pi}{8}$ above the horizon. Note that the overall score is minimized when one tower is hidden behind the other one.

3.4.2 Stochastic Sampling of Rule Parameters

A grammar typically comes with several parameters to steer the generation of the procedural 3D model. In CityEngine, grammars are encoded using the shape grammar language CGA [MWH*06], and continuous or discrete parameters can be defined as shown on the left in Fig. 3.4. Note that every parameter needs to be initialized with a default value. Authors can also define the range of meaningful values for parameters using `@Range` annotations. In CityEngine, these ranges are used for constructing the sliders in the user interface (Fig. 3.4 on the right) and for us they limit the space within which we generate samples.

A grammar contains a start rule which is applied on an initial shape which is annotated with `@StartRule` in CGA. But the initial shape is undefined and could be of arbitrary form, dimension, or geometry. Thus, it could have a high impact on the generation, e.g., a conditional rule might determine to build a high-rise building instead of a small house based on the size of the initial shape. As a consequence, we extend CGA with the

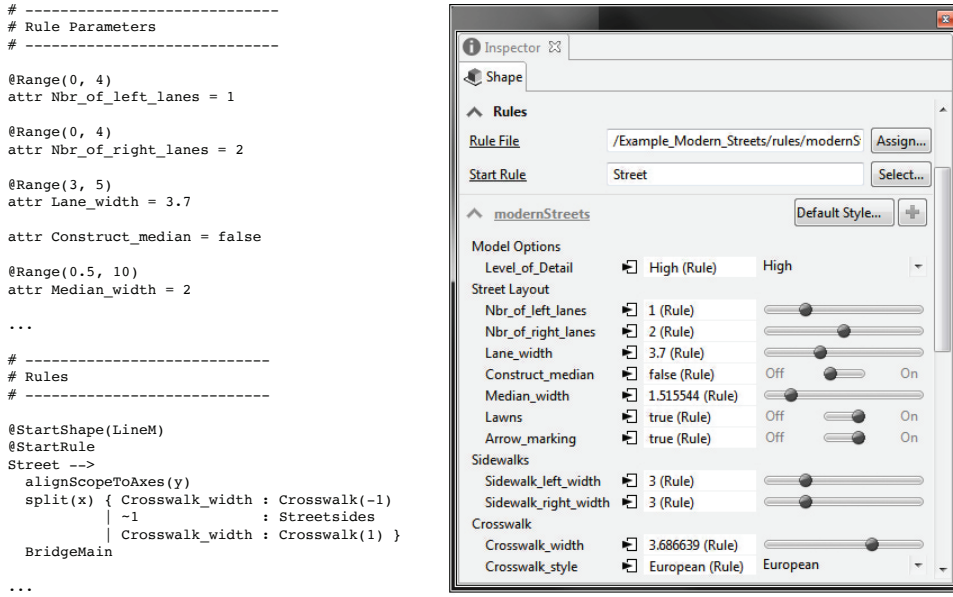


Figure 3.4: *Parameter Ranges and User Interface (left) CGA grammar excerpt with parameter ranges and start rule. (right) User interface to control the many parameters of the grammar.*

`@StartShape` annotation to set one or many predefined initial shapes such as `Point`, `Line`, `Rect` or `Cube`. In the example on the left of Fig. 3.4, the author uses `@StartShape(LineM)` to define the default initial shape as a line (the letters S, M and L denote dimensions 10, 30 and 100 meters).

As described in Alg. 3.1, we now stochastically sample the grammar within its given parameter ranges and initial shapes (by using CityEngine’s Python interface). For each resulting model we render the thumbnail image and calculate the view attributes from the best viewpoint found in Sec. 3.4.1. The reasons why we use only the best view of the default model even though it might not be the best view when jointly considering all models are twofold: 1) performance, i.e., calculating best views of more samples requires additional processing time, and 2) since all initial shapes are similarly aligned, changing the viewpoint for each sample only worsens the visual understanding of model differences. As an alternative we also experimented with calculating the best view of every sample and taking the one viewpoint that had the most support (every model sample would give one vote to its best viewpoint). The results are similar but there is the drawback of having to compute the view attributes for all viewpoints for all samples.

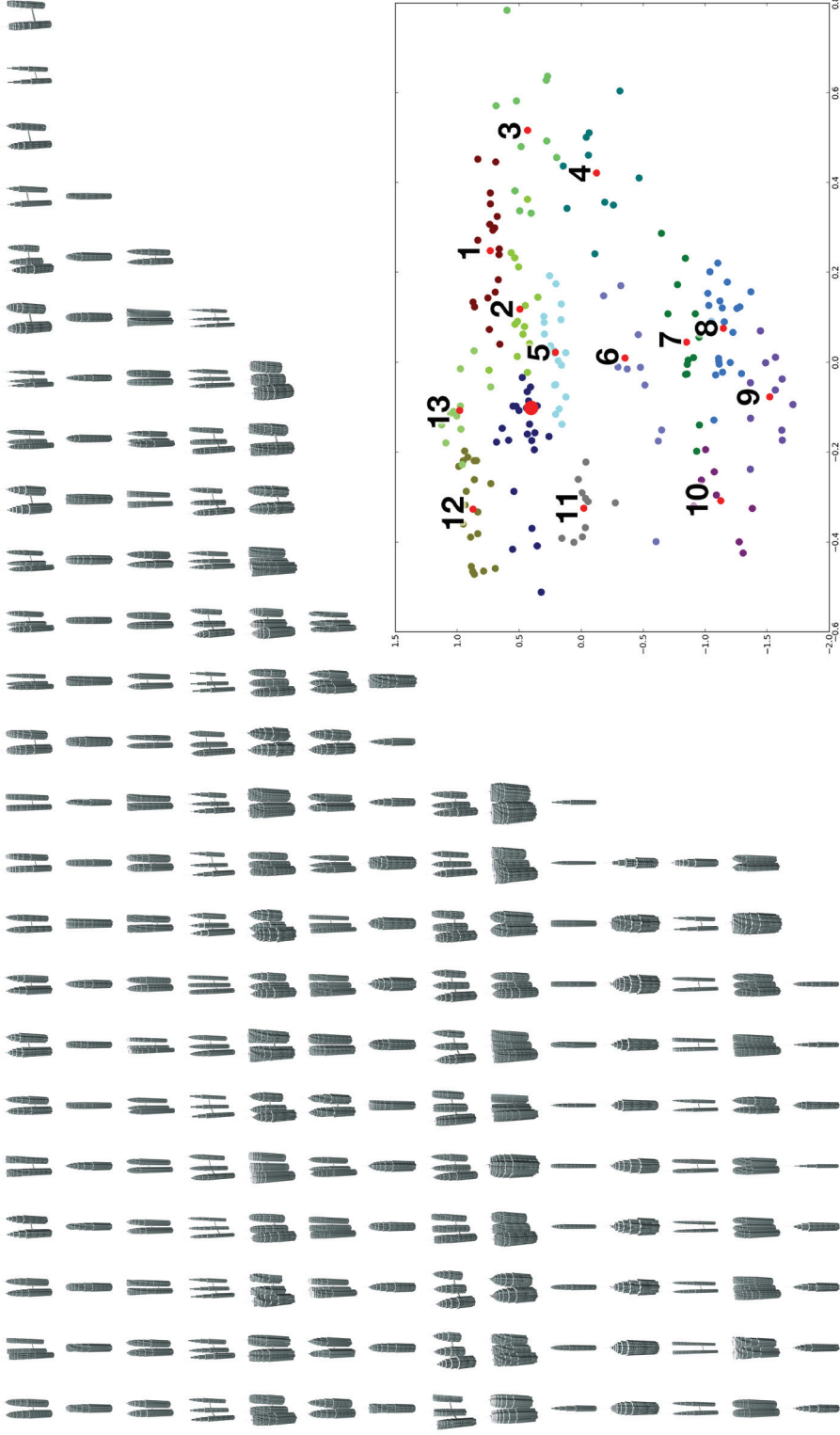


Figure 3.5: Clustering the Samples of the Petronas Towers Grammar. (left) Clustering results where each row of thumbnails represents one group with similar models. The cluster which received the most samples is depicted in the top row. The distance to each cluster center increases from left to right, i.e., the model on the left is the one nearest to the center of its cluster. (right) Representation of the clustering results using the first two principal components. The samples nearest to their cluster are marked red and the large one is the default model. The numbers illustrate the radial sorting around the default model and correspond to the numbers in Fig. 3.6.

3.4.3 Clustering View Attributes

As a next step, the list of view attributes is clustered into a given number of groups. Therefore we applied a slightly modified version of the k-means++ clustering algorithm [AV07]. K-means++ chooses only the first initial clustering seed completely at random and applies a distance-based probability heuristic to determine the remaining initial cluster seeds. However, to make the clustering even more stable, we set the first initial seed to the view attributes of the default model. This is justified by the fact that most rule authors intuitively place the default model near the center of the design space.

The cluster distance function is a linear combination of the view attribute deltas. Since our view attributes are typically well distributed in the $[0, 1]$ range, the weights reflect the importance of the corresponding view attributes for clustering. The weights we use were determined empirically, they are listed in Tab. 3.2 on the right. They provide visually appealing and representative clusters for arbitrary grammars. A cluster result is shown on the left Fig. 3.5.

3.4.4 Thumbnail Gallery Creation

To illustrate a cluster, we select the sample nearest to its center. To arrange the selected thumbnail images in a visually comprehensible way, we applied user interface principles of Design Galleries [MAB*97]. There, the current design is in the middle and suggested variations are arranged around it in a manner that correlates to the editing distances. In our case, the default model is the center and the selected thumbnails of the other clusters are arranged around it as shown in Fig. 3.6.

The leftmost thumbnail column in Fig. 3.5 shows that the cluster size is not a well-suited sorting criteria for the radial arrangement. The user wants to visually compare similar thumbnails and is less interested in the cluster sizes. Therefore, we run PCA on the view attributes to project the selected samples from their original eight-dimensional space into a two-dimensional subspace spanned by the first two principal components. Within this space, the representatives are sorted radially around the default model. This is illustrated in Fig. 3.5 on the right.

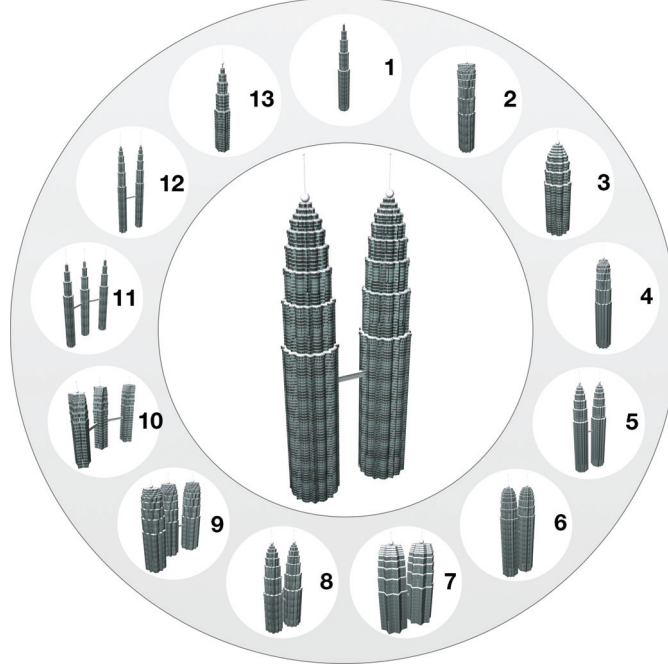


Figure 3.6: *Petronas Towers Thumbnail Gallery* The result of the *Petronas Towers* grammar. The thumbnail in the middle shows the default model and those around it depict other representatives. The ordering is the same as in Fig. 3.5.

This arrangement is also the reason why we typically generate 14 clusters. If we display the selected samples at third the size of the default model, we can fit exactly 13 thumbnails on the circle. Nonetheless, the number of clusters can be chosen by the user.

3.5 Results

3.5.1 Best View

To evaluate our best view method, we conducted a preliminary user study with 39 participants. The test data set contained 21 buildings, each rendered from the optimal perspective according to our view attribute combination and according to the linear-5 combination suggested by Secord et al. [SLF*11]. For our test set, both systems provide the same thumbnails for five of the exemplars (different view directions indicated with circles in Fig. 3.7). For the remaining models the subjects preferred our view direction in 351 cases, 161 views according to Secord et al., and had no preference in 112 cases. A significance test clearly rejects the hypothesis H_0 (H_0 : no preference, H_1 : preference

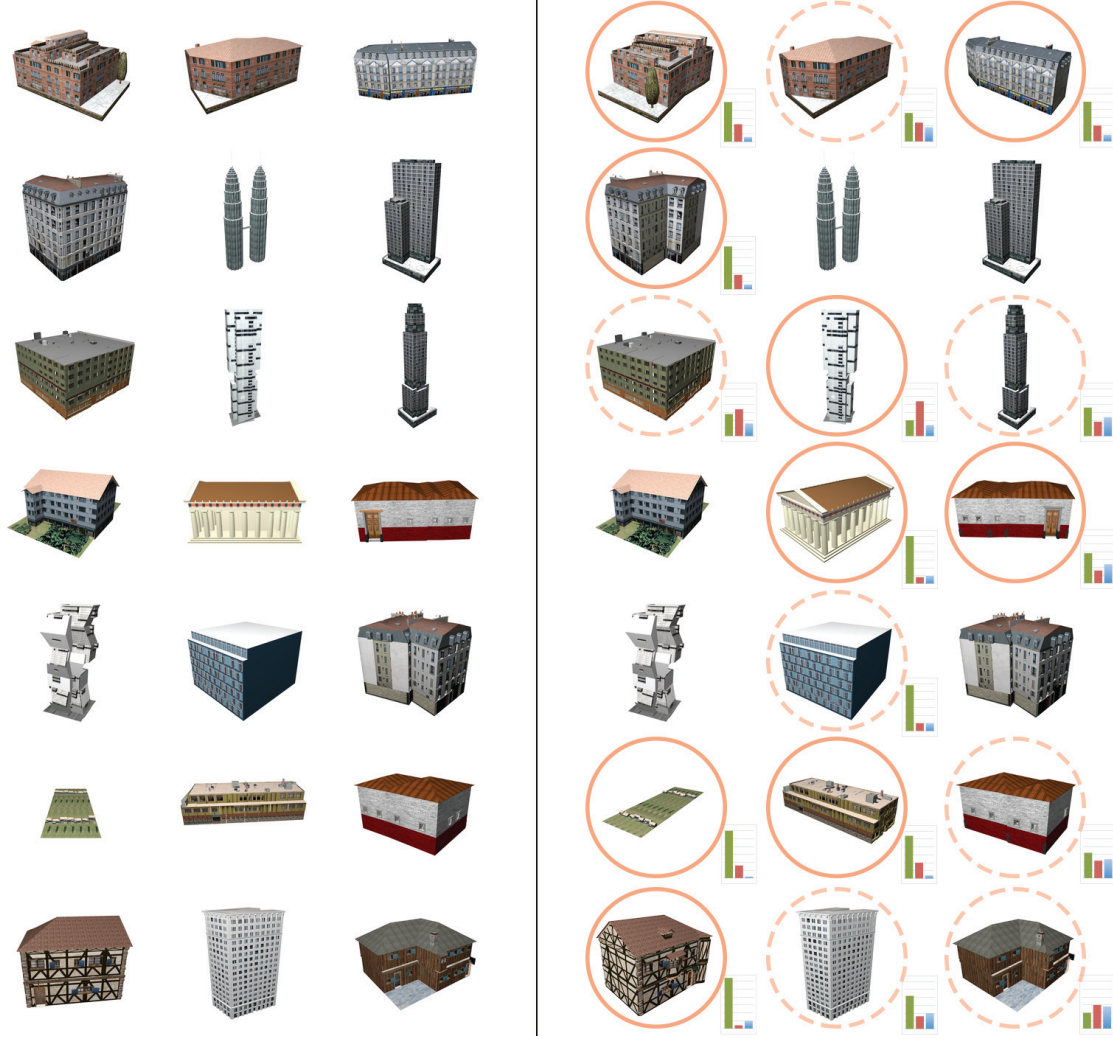


Figure 3.7: *Comparison with Secord et al. [SLF*11]* The results of Secord et al.’s view attribute combination are shown on the left and ours on the right. Circles denote differences between both methods, dashed circles stand for minor differences only. The bar charts show the user preference counts: left is ours, middle is Secord et al., right is no preference.

for the view direction obtained from our view attributes) within the $p < 0.001$ (one-tailed) confidence interval. Focusing on the models for which the two viewing directions considerably changed (more than 10 degrees, solid circles in Fig. 3.7) the subjects preferred our suggestions even more clearly (in 225 cases compared to 84 cases for the alternative, with 42 occasions of no preference).

Grammar	Model Generation	Best View	Attribute Calculation
Petronas Towers	334.00 s	0.43 s	64.34 s
Philadelphia	279.60 s	0.61 s	123.93 s
Paris	38.45 s	0.67 s	31.88 s
Procedural streets	113.27 s	0.35 s	111.08 s

Table 3.3: *Measurements* Running times for our system for different grammars using 200 samples. The best view selection algorithm considered 32 different viewpoints.

3.5.2 Clustering

Clustering behavior varies for different grammars. Generally, we observed that the clusters stabilize when at least 200 samples are used. K-means++ proved to be the most natural way to initialize clustering for our domain as we want the default model as the global center. We also experimented with other clustering methods with worse results: hierarchical clustering with dendrograms, spectral clustering, and mean shift clustering.

Fig. 3.8 shows thumbnail galleries for four different grammars: Philadelphia, Paris, Zurich, and procedural streets. The running times for these grammars are listed in Tab. 3.3. We used OpenGL and rendered the thumbnails at a resolution of 500×500 pixels on an Intel Core2 Duo 2.8 GHz Laptop with a Nvidia Quadro FX3700M graphics card. Clustering was done with NumPy and took about 0.5 s for each grammar. The times show that model generation is the most expensive part of the system. The attribute computation varies between grammars (Philadelphia takes four times longer than Paris). We believe that the reason for this is the increased number of lookup table reads (the number depends on the number of terminals and faces in the model) in former example.

3.6 Discussion and Future Work

Thumbnail galleries provide a new visual tool for procedural modeling that can significantly simplify rule selection during the content creation process. Our experiments demonstrate that the thumbnails automatically generated by our method yield good visual representations of the model diversity of a grammar.

Our system also has some limitations which open areas for future research. We currently fail to distinguish models that have a very similar overall shape that differs only on lower-scale structural details. We believe that view attributes encoding the silhouette (e.g., Fourier descriptors [ZR72]) or the shape of the 2D rendering (e.g., Zernike



Figure 3.8: Thumbnail Galleries The results for several grammars. Philadelphia (top left), Paris (top right), Zurich (bottom left), procedural street (bottom right).

moments [KH90]) could remedy those shortcomings. Another problem are rule parameters that change minor details, e.g., a building’s window width or the leaf shape of a tree. Those changes are hard to spot from a perspective that features the full model. A beneficial feature of the system would be to detect those shapes and present close-up images. One could further investigate new color and texture based view attributes as our system ignores this information almost entirely (color changes influence the luminance images and have a minor effect on the contrast view attribute). Fig. 3.9 shows a problem case where a grammar for futuristic high-rise buildings cannot be clustered properly.



Figure 3.9: *Clustering Problem* All these high-rise buildings from the same grammar were mistakenly placed in the same cluster because they have a similar overall shape and size. Additional view attributes, encoding color and silhouette shape, could fix such problems.

User-guided exploration of the procedural design space spanned by the grammar similar to Design Galleries [MAB*97] is also worth exploring: the user would repeatedly select a model while the system automatically provides new suggestions of models close to the selected one. A related question is if it is possible to reverse-engineer the influence of rule parameters given the view attribute.

Our system currently just samples parameters which have an optional range annotation. User-guided refinement could be used to detect sensible ranges for non-annotated parameters. Also our sampling strategy could be improved with an adaptive approach. Rather than sampling all parameters with the same probability we could detect the ones which lead to large variations of the view attributes and sample them more densely.

3.7 Conclusions

We present a system that finds the best view of a procedural model and that generates a thumbnail gallery depicting the design possibilities of a grammar. The best view selection algorithm works well for arbitrary mesh topologies and outperforms existing methods. Clustering in conjunction with our novel view attributes is a first step towards visualizing the immense variety of models encoded in a procedural grammar. To the best of our knowledge, we are the first to present such a system.

4 Designing Probability Density Functions for Shape Grammars

We introduce a novel framework that allows designing a probability density function (pdf) for a shape space spanned by a stochastic grammar and that adapts the grammar to generate new models according to the designed pdf. The user assigns preference scores to a set of exemplar models sampled from the grammar. A kernel function that compares the shape trees of two exemplar models is used in conjunction with Gaussian process regression (GPR) and automatic relevance determination to interpolate the preference scores over the entire shape space. In order to enable model generation according to a pdf proportional to the learned preference function, the original grammar is changed by applying split operations to relevant non-terminal symbols and by modifying the rule probabilities. Our framework is compared to state-of-the-art exploratory modeling techniques, and we evaluate it using different design tasks for buildings, furniture, trees, and airplanes.

4.1 Introduction

In procedural modeling, a stochastic shape grammar defines a shape space containing a variety of models of the same class, e.g., vegetation or architecture. Users of procedural modeling tools such as CityEngine can be broadly divided into two groups: 1) non-experts without programming experience coming from such fields as architecture, urban planning, or arts, and 2) experts with a programming background, some with and some without artistic talent. Authoring shape grammars is a non-trivial task as it requires programmings skills. Hence, usually only expert users model new shape grammars from scratch while non-experts mostly work with existing grammars, sometimes modifying them slightly to meet their needs. Both groups are confronted with the common problem

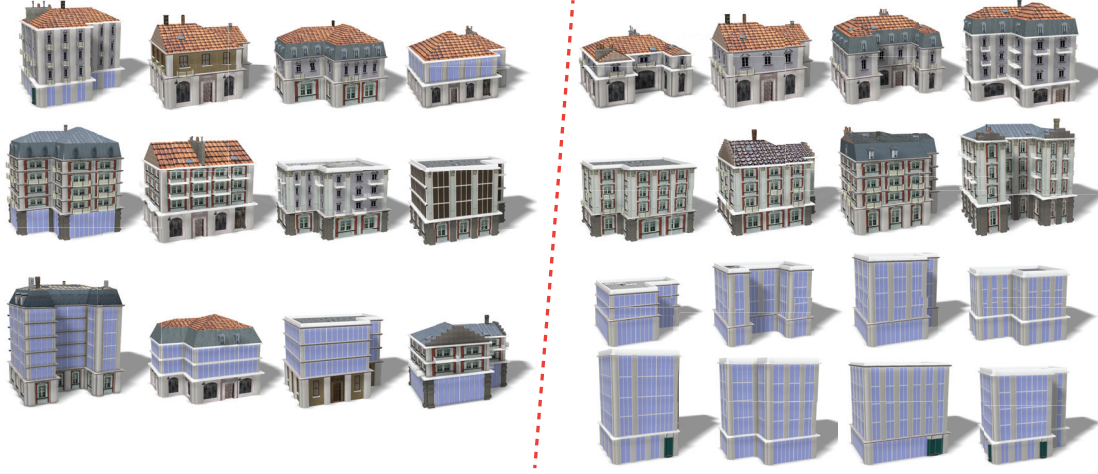


Figure 4.1: Framework Overview (left) A selection of random models sampled from a stochastic shape grammar for buildings. Even though one can physically build them, they do not make sense architecturally: the styles of the ground floors, upper floors, and the roofs mismatch. (right) Given user-provided preference scores, our framework learns a new probability density function (pdf) and adapts the stochastic grammar to generate new models (consistent in style) in proportion to the pdf. This design task (B1 in Tab. 4.2) assigns a higher preference score to office buildings than to the two different types of residential buildings.

that it is difficult to control the distribution of all the models in a shape space of a grammar. The pdf of a grammar defines for each model how probable it is that it will be generated. With our novel framework users can interactively design a new pdf for an existing grammar. Our system works without requiring the user to understand or to directly edit any of the grammar’s production rules. In the following we describe three common procedural modeling design tasks that form the motivation for our framework.

In the first task, a non-expert user searches for just one interesting model within a grammar’s shape space. The common solution is to naively keep regenerating sample models with new random seeds until a suitable one is found. This will likely take a long time, especially if the user is picky and if the shape space is large, covering a wide range of variations. Building on existing exploratory modeling methods [MAB*97, TGY*09], we enable interactive exploration of the shape space by designing a pdf that has a high density near previously liked samples.

A second task might require the generation of many procedural models, e.g., enough building models to fill a city. The user would like to have control over the distribution of the size, color, or type of the buildings. While it can be trivial (for expert users) to change the grammar directly to enforce a distribution for just one such parameter, it

becomes more difficult when several interdependent parameters are involved, e.g., when residential buildings should be short while a high percentage of tall buildings should only contain offices.

Third, when a procedural grammar is very general, i.e., has a wide range of variations, it is very difficult to guarantee that all possible combinations of stochastic rules yield meaningful models that correspond to the author’s intent. As the grammar becomes more general, the overall quality of the models usually degrades because the number of unwanted models increases. For example, a very generic plant grammar could generate a tree with impossible branching angles because it combines a tree species rule with an incompatible branching rule. Again, such constraints are hard to encode directly in grammars, even for expert users. This is because the rules are context-free and do not know any global context. The third task, similar to the second task, wants to modify the likeliness of certain models; it wants to avoid undesirable models completely by minimizing their probability. With our tool, such constraints can be enforced through adaptation of the pdf and do not require grammar authors to write overly complicated grammars. Fig. 4.1 shows an example combination of tasks two and three. It avoids undesirable models and controls the distribution of the remaining ones.

To summarize, we present the following main contributions. On the application side, we extend exploratory modeling to not only find specific models within a shape space but also to designing a pdf over the entire shape space. On the technical side, we propose a kernel function that, given two derivations from a stochastic grammar, can measure how similar they are. We use function factorization and GPR (with automatic relevance determination and a lasso regularizer) to actively learn a preference function from a set of user-scored exemplar models. We further provide an algorithm that adapts a grammar to generate models according to a learned target pdf, and we also show that our learning algorithm converges faster than Talton et al.’s method for kernel density estimation [TGY*09].

4.2 Related Work on Exploratory Modeling

The foundations of shape grammars are described in Chap. 2. In this section we solely give a concise overview of the most important design exploration methods. We have recently seen several research efforts for exploratory modeling in the graphics community. The first one was the concept of Design Galleries by Marks et al. [MAB*97] enabling navigation of parameter spaces, e.g., for light placement in rendering and for motion

control in animations.

Certain methods investigate the exploration of already existing, discrete spaces of designs, e.g., of sets of websites [LSK*10] or 3D models [KFLCO13]. Even crowdsourcing has been used to learn and compare high-level descriptive attributes for designs [CKGF13, OLAH14]. In another project, Averkiou et al. [AKZM14] embed shapes in low-dimensional spaces that can be explored, and an inverse mapping is used to synthesize new shapes.

Usually design spaces are explored by showing the user samples that are similar to other, previously liked samples. This applies to a wide range of generative models, e.g., probabilistic [MSL*11] or evolutionary [XZCOC12] ones. Sometimes the shape spaces are constrained and the exploration needs to be interleaved with optimization steps to guarantee valid or good models [UIM12, BYMW13]. This is often the case for free-from surfaces in architectural modeling [YYPM11, DBD*13]. Also, in Chap. 3 we presented a strategy to create concise previews of shape spaces of procedural grammars [LSN*14].

Other methods explore parametric models. For example, Brochu et al. [BBdF10] use Bayesian optimization to navigate animation parameters, Shapira et al. [SSCO09] recolor images based on Gaussian mixture models, and Koyama et al. [KSI14] learn goodness values for samples via crowdsourcing and use radial basis functions to generate new samples of interest by interpolation. Most similar to our framework is Talton et al.'s [TGY*09] method which uses kernel density estimation to learn the distribution of desirable models from user input and map it into an explorable 2D space. For parametric models it is straight-forward to generate new designs by directly sampling from the distribution. We, on the other hand, work with grammars and have no explicit parametrization of the shape space. We additionally have the challenge of adapting grammars so that their output corresponds to a given pdf.

In comparison to all these methods, our project's main focus is not on exploration but on modeling the output of a generative process by means of a pdf. Exploring shape spaces is possible with our framework, but more importantly it can also adapt their probability distributions.

4.3 Overview

4.3.1 Framework Overview

Our system takes a stochastic grammar as input. The default production rule probabilities define a *default pdf* for the shape space. Deriving the grammar generates output models that are distributed according to that default pdf. The objective of our framework is to learn a new pdf from user-provided preference scores and to modify the grammar and its rule probabilities to generate new models according to the new pdf.

Our system consists of three main parts: 1) a GPR module that learns a preference function from models scored by the user (Sec. 4.4), 2) a grammar adaptation algorithm that modifies the probability distribution of the grammar (Sec. 4.5), and 3) a graphical user interface that iteratively refines a preference function by displaying sets of sample models that the user can score on a scale of $[0, 100]$ (Sec. 4.6).

4.3.2 Grammar Definitions

We work with stochastic, context-free shape grammars that are compatible with CityEngine. Shapes have labels, either non-terminals $\in NT$ or terminals $\in T$, and such attributes as a *scope*, a mesh tightly enclosed by the scope, and a material specification. Throughout this chapter, we use the *toy grammar* in Fig. 4.2 as an example to illustrate several mechanisms. Random samples from the toy grammar’s default pdf are shown in Fig. 4.3. We define a shape grammar slightly different than in Chap. 2, namely as:

$$G = \langle NT, T, \omega, P, \Theta \rangle,$$

where $\omega \in NT$ is the *axiom*. The production rules P are of the following form:

$$id_i \text{ predecessor} \xrightarrow{p} \text{successor}_i$$

Each rule can apply only to a shapes labeled with *predecessor*, and it is used with probability p . Every *successor_i* is a procedure that applies shape operations (e.g., transformations, splits, material assignments, etc.) to the predecessor shape and replaces it with a set of child or successor shapes ($child_{i1}, \dots, child_{ik}$), where the label of $child_{ij}$ is $\in NT \cup T$. We will use the *child index* $j \in [1, k]$ later on for identification of individual child shapes. Rules are additionally enumerated with identifiers id_i to distinguish stochastic choices.

NT	$=$	$\{ \omega, Mass, Mesh, ColorMesh \}$
T	$=$	$\{ TerminalMesh \}$
attr h_1	$=$	<code>rand_uniform(1, 5)</code>
attr h_2	$=$	<code>rand_uniform(1, 5)</code>
id_1	ω	$\xrightarrow{1} \text{split}(y) \{ h_1 : Mass \mid h_2 : Mass \}$
id_2	$Mass$	$\xrightarrow{0.\bar{3}} \text{i}(\text{"box"}) Mesh$
id_3	$Mass$	$\xrightarrow{0.\bar{3}} \text{i}(\text{"cylinder"}) Mesh$
id_4	$Mass$	$\xrightarrow{0.\bar{3}} \text{i}(\text{"star"}) Mesh$
id_5	$Mesh$	$\xrightarrow{1} ColorMesh$
id_6	$ColorMesh$	$\xrightarrow{0.\bar{3}} \text{setColor}(\text{"brown"}) TerminalMesh$
id_7	$ColorMesh$	$\xrightarrow{0.\bar{3}} \text{setColor}(\text{"blue"}) TerminalMesh$
id_8	$ColorMesh$	$\xrightarrow{0.\bar{3}} \text{setColor}(\text{"turquoise"}) TerminalMesh$

Figure 4.2: Toy Grammar This very simple grammar generates two shapes that are stacked on top of each other. The appearance of every shape is defined by three random choices for height, asset mesh (box, cylinder, or star), and color (brown, blue, or turquoise).

The set of parameters Θ is also new in G . It contains all rule probabilities and all parameters of any random number generators used during derivation. For example, the heights h_1 and h_2 in the toy grammar are randomly sampled from uniform distributions whose ranges are included in Θ .

The toy grammar's first rule id_1 stacks two generic shapes with random heights (h_1 and h_2) on top of each other. The type of each shape is decided in the rules id_2 , id_3 , id_4 that chose between a box, cylinder, or star (using the **i** operation to instantiate meshes) with $\frac{1}{3}$ probability. Rules id_6 , id_7 , id_8 define the shape color to be either brown, blue, or turquoise, also with uniform probability. Rule id_5 does not offer any stochastic choices and is therefore redundant for our purposes. This simple toy grammar highlights some challenges for modeling arbitrary pdfs with context-free grammars. For example, it is impossible to change the variables in Θ such that the top shape is always blue and the bottom shape has a random color. This can only be achieved when id_2 through id_8 are duplicated and modified to assign different color probabilities for each shape. If we added more shapes to the stack, the encoding of specific combinations of colors and shapes would explode combinatorially, and the grammar would become harder to understand. To define a generic joint pdf over several variables, one needs an exponential number of production rules in the grammar. Our framework can do this adaptation of the grammar

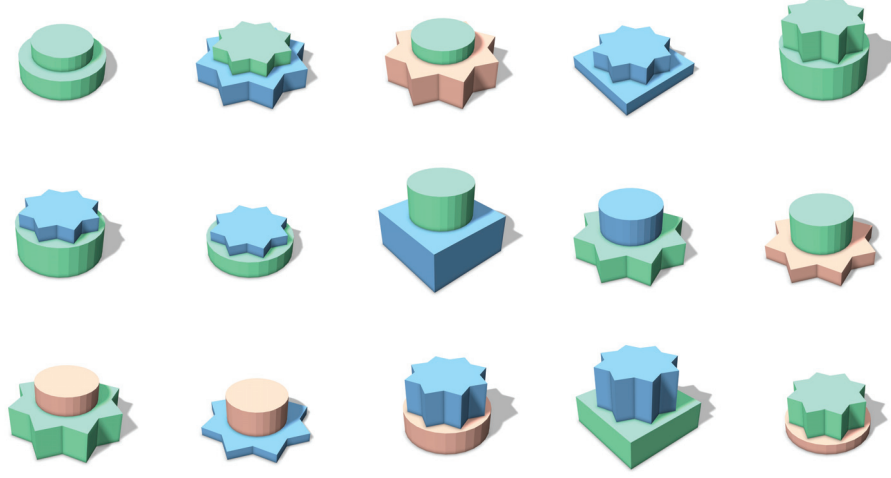


Figure 4.3: Toy Grammar Samples Random models sampled from the toy grammar defined in Fig. 4.2 and described in Sec. 4.3.2.

automatically behind the curtain without requiring the user to touch or look at the grammar code.

4.4 Learning the Probability Density Function

There are two main steps for learning a new pdf. First, we define a function that maps models of a grammar to features vectors (Sec. 4.4.1). We assume that preference scores vary smoothly over that feature space. Second, we explain how we model a preference function as a Gaussian process to interpolate preference scores from a few samples (that the user provides through the interface explained in Sec. 4.6) over the remaining parts of the shape space (Sec. 4.4.2). We describe the regression, the regularization, and also how to find optimal hyperparameters for the kernel function. A new pdf is then obtained by normalizing the preference function learned with GPR. There is also a third optional step that allows creating a global preference function by combining several individual ones as factors (Sec. 4.4.3).

4.4.1 Features

Finding a good mapping function from a rule-based model m to the feature space \mathcal{X} is difficult. Geometric features are not optimal because different grammars require separate

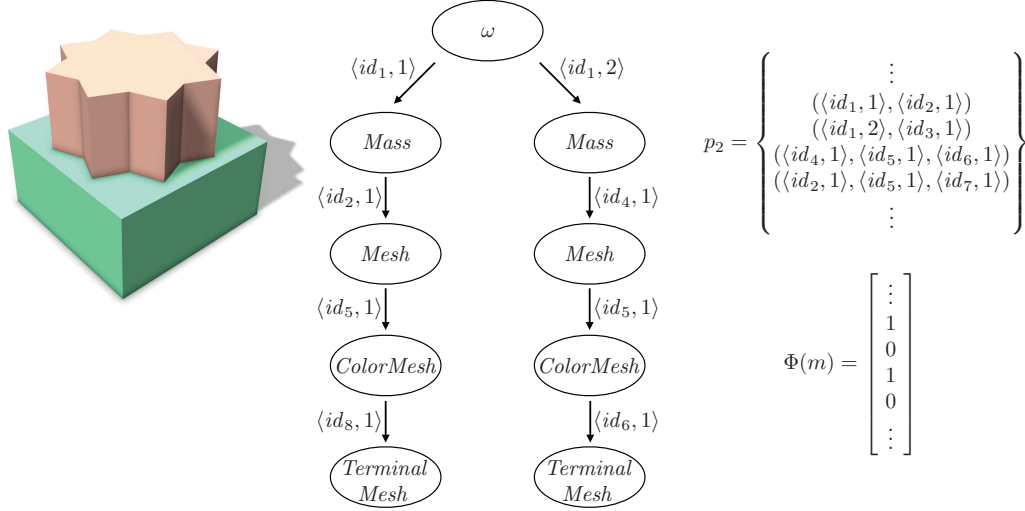


Figure 4.4: Features A random model m derived from the toy grammar is displayed together with its shape tree. The set p_2 contains all shape tree paths of effective length two. Each shape tree path is mapped into one component of the feature vector $\Phi(m)$. The mapping function simply counts how many times a certain path exists in the shape tree.

features designed specifically for the grammar. The image-based view attributes from Chap. 3 are unfortunately not discriminative enough for our design tasks. For parametric models the parameters themselves can often be used directly as features, e.g., Talton et al. [TGY*09] do that. Grammar parameters have been used as features before [STKP11] but only for simple façade grammars with a narrow shape space of similar models. We use the information encoded in the shape trees of the rule-based models to define a set of features with sufficient discriminative power to compare models. An example shape tree is shown in Fig. 4.4. The nodes of the shape tree represent shapes, and they are labeled by the corresponding shape labels, i.e., terminal and non-terminal symbols of the grammar. The edges of the tree represent rules labeled by a tuple consisting of the rule id and the child index. The child index consistently numbers all shapes generated by the rule.

We can identify a *path* in the shape tree by the sequence of labels on its edges. For example, for the shape tree in Fig. 4.4 we can observe a path $(\langle id_1, 2 \rangle, \langle id_4, 1 \rangle)$. We only consider paths that follow the order of derivation, i.e., go from parents to children. We can observe that the paths encode the relationships between shapes and shape attributes. For example, if $(\langle id_4, 1 \rangle, \langle id_5, 1 \rangle, \langle id_6, 1 \rangle)$ exists, there exists an brown star in m (either at the top or the bottom), while $(\langle id_1, 1 \rangle, \langle id_2, 1 \rangle, \langle id_5, 1 \rangle, \langle id_8, 1 \rangle)$ indicates that the bottom shape is a turquoise box.

4.4. Learning the Probability Density Function

A model m generated by the grammar G is mapped into the feature space with a mapping function $\Phi(m)$ with the following components:

- The values θ_m assigned to all random parameters of shape operations in any *successor_i* used to derive m . (In a recursive grammar, we combine the parameters of all recursive rules up to a predefined level of recursion.)
- The number of times each symbol ($\in NT \cup T$) occurs in the shape tree.
- The number of times a path occurs in the shape tree.

The number of different paths in a shape tree can become very large even for shape trees of moderate size. Therefore, we limit the feature vector to a subset of paths that are useful. The ability to discriminate properties differs with the length of the path. Short paths can distinguish general properties, e.g., a brown box or a turquoise star. Longer paths identify more specific elements, e.g., a brown box on the bottom or a brown box on top. Since short paths are sufficient to identify simple concepts, we start with short paths of maximal length k . Once the current feature vector becomes unable to differentiate models with substantially different preference scores, we gradually increase k to include longer paths.

There are several details to consider. First, some rules have no discriminative information. For example, in the toy grammar, if rule id_4 is executed, the child shape will always be replaced with rule id_5 . We define the *effective length* of a path by discarding these non-informative rules. For example, the path $(\langle id_4, 1 \rangle, \langle id_5, 1 \rangle, \langle id_6, 1 \rangle)$ has length 3, but effective length 2 since id_5 is not counted. We categorize paths by their effective length instead of their length. Second, since superfluous features increase the chance of overfitting, we typically start with $k = 1$. The number of features D also defines the dimension of the feature space \mathcal{X} . In Fig. 4.4 we show the extraction of selected features from a shape tree.

4.4.2 Gaussian Process Regression (GPR)

GPR is a kernel-based Bayesian regression technique. In our case it imposes a Gaussian process as a prior for the preference function $u(m)$ that we want to learn, where m is a model derived from the given grammar. This means that a set of n models with scores $[u(m_1), \dots, u(m_n)]^T$ (assigned by the user) can be seen as an n -dimensional sample of an n -variate Gaussian distribution. GPR gives reasonable results even for small numbers of

observations, which allows our system to work iteratively. It already provides meaningful output for the first few observations, and it updates the preference function whenever new observations are available. For an in-depth review of Gaussian processes, we recommend the excellent review by Rasmussen and Williams [RW06].

A Gaussian probability distribution is characterized by a mean and covariance. Similarly, a Gaussian process is governed by a mean function and a kernel function $k(m, m')$ that, in our case, measures the similarity between the two derivations m and m' . We learn the hyperparameters of the kernel function directly from the user input. The mean function expresses a bias for the prior of u and is commonly chosen to be zero, which means that none of the models are desirable at the start. This has been proven successful in related contexts such as the ranking of songs [PBS*01].

Preference Prediction We have n models $\mathbf{m} = [m_1, m_2, \dots, m_n]^T$ and their associated preference scores $\mathbf{u} = [u(m_1), u(m_2), \dots, u(m_n)]^T$. Our objective is to make predictions about the preference $u(m^*)$ of another arbitrary model m^* .

Given a Gaussian process prior with zero mean for $u(m)$ and the kernel function $k(m, m')$, $[\mathbf{u}, u(m^*)]^T$ is described by an $(n + 1)$ -variate zero mean Gaussian distribution with the following covariance matrix:

$$\mathbf{C}_{n+1} = \begin{bmatrix} k(m_1, m_1) & \dots & k(m_1, m_n) & k(m_1, m^*) \\ & & \vdots & \\ k(m_n, m_1) & \dots & k(m_n, m_n) & k(m_n, m^*) \\ k(m^*, m_1) & \dots & k(m^*, m_n) & k(m^*, m^*) \end{bmatrix}.$$

A closed form solution for $u(m^*)$ is given by the conditional distribution $p(u(m^*)|\mathbf{u})$. It is also Gaussian and has a mean of:

$$u(m^*) = \mathbf{k}^T \mathbf{C}_n^{-1} \mathbf{u},$$

where \mathbf{C}_n is the $n \times n$ submatrix in the upper left of \mathbf{C}_{n+1} and

$$\mathbf{k} = \begin{bmatrix} k(m_1, m^*) \\ \vdots \\ k(m_n, m^*) \end{bmatrix}.$$

GPR not only predicts $u(m^*)$ but it also has the nice property of giving us a confidence

4.4. Learning the Probability Density Function

estimate of the prediction. This *prediction uncertainty* is the variance of $p(u(m^*)|\mathbf{u})$ and is defined as:

$$\sigma^2(m^*) = k(m^*, m^*) - \mathbf{k}^T \mathbf{C}_n^{-1} \mathbf{k}$$

Our user interface (Sec. 4.6) can use the prediction uncertainty to present models that it is not knowledgeable about. This will usually speed up the learning convergence because GPR gains more information from models with high uncertainty than from models with low uncertainty.

Kernel Function The type of kernel function used to interpolate the preference scores from a few exemplar models over the entire shape space influences the outcome of the GPR. We do not use a fixed kernel but rather choose a family of parameterized kernel functions for which we learn parameters from the user input.

The anisotropic kernel function k , used to compare two rule-based models m and m' in the D -dimensional feature space \mathcal{X} , is defined as:

$$k(m, m') = \theta_0 \exp \left(-\frac{1}{2} \sum_{d=1}^D \theta_d (x_d - x'_d)^2 \right) + \delta_{mm'} \beta^{-1},$$

where x_d and x'_d are the components of the feature vectors of m and m' , $\delta_{mm'}$ is the Kronecker delta, and β^{-1} is a small noise-like term that is added to the diagonal elements of the covariance matrix to guarantee its positive-definiteness. The weights $\theta_d, d \in [1, D]$ are the hyperparameters of the kernel, and they are not constant. They are readjusted every time the user scores new models, and they reflect the importance of their corresponding features. The importance of a feature typically depends on the design task and the preference function that the user is modeling. A large weight indicates that a feature has a strong influence on the preference scores.

Automatic Relevance Determination We use *Automatic Relevance Determination (ARD)* to learn the hyperparameters. ARD finds the set of hyperparameters $\boldsymbol{\theta} = (\theta_0, \theta_1, \dots, \theta_D)$ for the kernel function that maximize the log likelihood of the user-scored training models:

$$\ln p(\mathbf{u}|\boldsymbol{\theta}, \mathbf{m}) = -\frac{1}{2} \ln |\mathbf{C}_n| - \frac{1}{2} \mathbf{u}^T \mathbf{C}_n^{-1} \mathbf{u} - \frac{n}{2} \ln(2\pi).$$

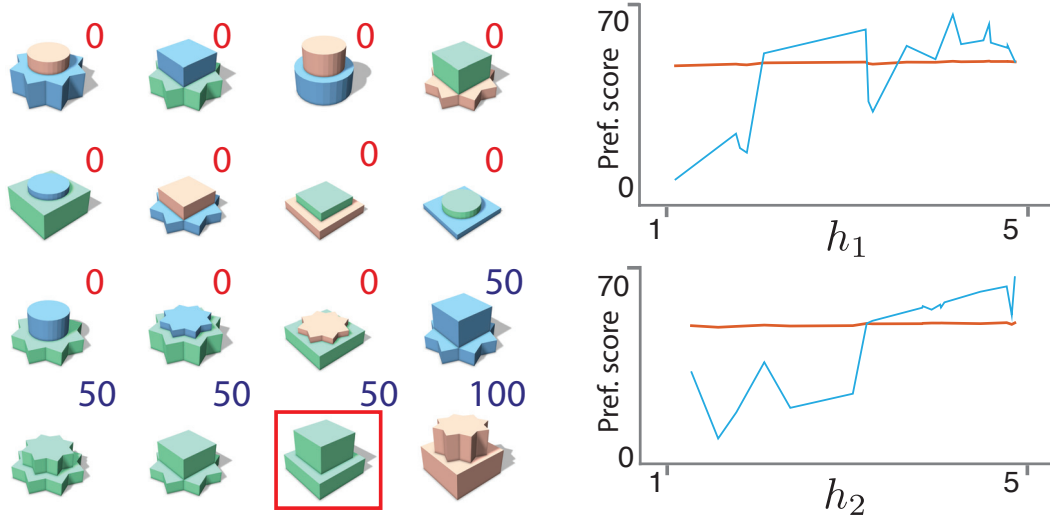


Figure 4.5: Automatic Relevance Determination (ARD) The left side shows preference scores assigned to 16 models of the toy grammar according to a modeling task that only allows designs that have the same color for both shapes (preference score 100 for brown and 50 for blue and turquoise). We then generate many variations of the highlighted model with two turquoise boxes by randomly changing the shape heights h_1 and h_2 within the range $[1, 5]$. Predicted preference scores for h_1 and h_2 are plotted on the right side, with ARD (orange) and without (blue). The almost constant orange curves for the different values of the heights indicate that ARD correctly detects the irrelevance of the height parameters for this specific task.

We add a lasso term for regularization that prevents overfitting. We use the conjugate gradient method to minimize the resulting energy function:

$$E(\boldsymbol{\theta}) = -\ln p(\mathbf{u}|\boldsymbol{\theta}, \mathbf{m}) + \lambda \sum_{d=1}^D |\theta_d|.$$

The hyperparameters θ_d should be positive, hence $|\theta_d| = \theta_d$. To avoid non-negativity constraints ($\theta_d > 0$) we optimize for $\ln(\theta_d)$. Additionally, the lasso regularizer has the advantage of driving as many hyperparameters as possible to zero. This way, only relevant features receive a non-zero weight, and the other features are discarded.

The effect of ARD is demonstrated in Fig. 4.5. The user wants to design a preference function for which only colors are relevant. Designs should have the same color for the top and bottom shapes, with a preference of 50 for blue and turquoise and 100 for brown. The shape heights h_1 and h_2 have no influence, and ARD can detect that.

4.4.3 Preference Function Factorization

Our framework allows modeling either a single preference function, or one can model an overall preference function as a product of K *factors*, i.e., K individual preference functions multiplied together:

$$u(m) = \prod_{i=1}^K u^i(m), \quad (4.1)$$

where K is chosen by the user. Having factors allows specifying the preference for particular aspects individually, e.g., for materials, window types, or roof shapes. It is often easier to set preference scores for individual aspects than for many aspects at once.

A design task for the toy grammar could have two factors: 1) *color* that assigns scores 100 to two brown shapes, 50 to two blue or two turquoise shapes, and 0 to the rest, and 2) *geometry* that assigns scores based on the type of the bottom shape, i.e., 100 for a box, 50 for a cylinder, and 0 for a star. Fig. 4.6 presents examples of both factors and their combination. Commonly, each factor has different relevant features, and thus one can often learn factors more efficiently than a global preference function. Each factor is learned individually with our user interface.

4.5 Generating Models According to a PDF

After training a new preference function $u(m)$ for the grammar we want to generate new models sampled according to the pdf that is proportional to the preference function. We experimented with three different ways for generating samples: 1) *rejection sampling*, 2) *parameter learning* that changes production rule probabilities, and 3) *structure learning* that modifies the grammar structure by adding new rules in addition to changing the rule probabilities.

Rejection sampling outputs models that are distributed exactly according to the designed pdf. However, it can be very inefficient because many samples might be discarded. Parameter learning is faster as no samples are discarded, but it is inflexible; just changing rule probabilities is usually insufficient to approximate the target pdf well. Our default solution is structure learning that approximates the pdf well, and it can generate new samples without rejection after an initialization preprocess.


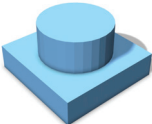


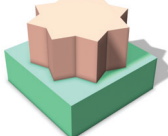
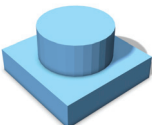


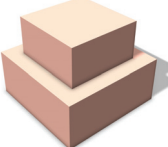
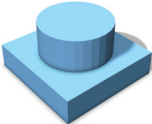


Factor 1	 100	 50	 50	 0
Factor 2	 100	 100	 50	 0
Combination	 100	 50	 25	 0

Figure 4.6: Preference Function Factorization The preference function can be learned as a combination of factors. Two factors for the toy grammar for color and geometry (see Sec. 4.4.3 for details) are learned separately and combined into an overall preference function.

Rejection Sampling This approach does not change any aspects of the grammar. It samples from the grammar’s default pdf and discards unwanted samples. The probability $p(m)$ that the grammar generates a specific model m (according to the default pdf) is the product of the probabilities of all the rules used to create m and also of all involved parameters (e.g., the heights h_1 and h_2 in the toy grammar). The new sample model m is accepted only with probability $\frac{u(m)}{Mp(m)}$, where M is the upper bound of $\frac{u(m)}{p(m)}$. If it is rejected instead, m is resampled as many times as necessary until it is accepted. We derive M empirically from a large number of samples.

Parameter Learning This algorithm leaves the grammar structure intact and only adapts the production rule probabilities to maximize the chance of generating a specific set of training models. The rule probabilities that assign maximum likelihood to the training set are given by a relative frequency estimator [Sto94, Joh98]. The optimal probability $\hat{p}(id_x)$ for a rule id_x , with respect to the training set, is the ratio of the number of uses of id_x and the number of uses of rules with the same predecessor as id_x

(including id_x itself):

$$\hat{p}(id_x) = \frac{c(id_x)}{\sum_{id_i \in \{\text{rules with same predecessor as } id_x\}} c(id_i)},$$

where c counts how often a rule is applied for deriving the training models. We obtain a sufficiently large set of training models from the original grammar by rejection sampling. Parameter learning is too simple and performs badly as shown in Tab. 4.3. The default structure of a grammar is often not suited to encode design tasks that have dependencies between choices of production rules. For example, parameter learning cannot adjust the default toy grammar to assign blue colors to star shapes with higher probability than to other shapes. It can only modify the color probabilities for all shapes at once.

Structure Learning Our preferred solution overcomes the drawbacks of both previous sampling strategies in two steps. 1) We first generate a new grammar adapted to the relevant features by applying *split* operations to duplicate relevant rules in the original grammar. We can then adjust the probabilities of the duplicated rules individually and gain enough flexibility to describe interdependencies of production rules. 2) We define the final grammar as a mixture of several grammars. Each component grammar of the mixture has the same structure as the adapted grammar resulting from the first step, but with different rule probabilities so that each component can focus on different aspects of the pdf.

The inspiration for the split operation is based on the idea of parent annotation in natural language processing [Joh98] and a similar splitting concept introduced by Talton et al. [TGY*09]. A non-terminal symbol S can be split into multiple new non-terminals. Every occurrence of S in a successor of any production rule will be replaced by a symbol S_i , where i enumerates the occurrences. The production rule for S has to be duplicated for each S_i . The successors of all the production rules for S_i are identical, but the rule probabilities can differ. For example, splitting the *Mass* symbol in the toy grammar leads to different symbols for the top and bottom shapes after deriving the axiom. The consequence is that the probabilities of the shape types (box, cylinder, star) become independent for the bottom and the top. It is important to note that splitting a non-terminal symbol does not alter the shape space of a grammar. But it does add more degrees of freedom to adjust the probability distribution of the grammar. Splitting all non-terminal symbols in a grammar increases the number of rules exponentially. Therefore we split only those symbols that occur in the relevant features that were

assigned a non-zero weight during ARD (Sec. 4.4.2).

The second step defines the final grammar G' as the mixture of K component grammars G'_i . Each component is selected with its corresponding probability α_i :

$$G' \xrightarrow{\alpha_i} G'_i.$$

Given the set of training models $\mathbf{T} = [m_1, m_2, \dots, m_N]^T$, we want to find K and also the hyperparameters $\boldsymbol{\eta}$ of the final grammar that maximize the log likelihood of the training models $\mathcal{L}(\mathbf{T}|G') = \sum_{i=1}^N \ln p(m_i|G')$. The hyperparameters $\boldsymbol{\eta}$ consist of all α_i and all grammar parameters and rule probabilities of each component grammar G'_i . The probability $p(m|G')$ that the final grammar derives a specific model m is a weighted average of the probabilities of the component grammars G'_i :

$$p(m|G') = \sum_{i=1}^K \alpha_i p(m|G'_i).$$

Our optimization for $\boldsymbol{\eta}$ is almost identical to the expectation–maximization (EM) algorithm for Gaussian mixture models. The difference is that the M-step changes the parameters of the component grammars and not of Gaussians. The E-step computes the *responsibilities* w_{kn} that measure how likely a model m_n is to be generated by component grammar G'_k :

$$w_{kn} = p(G'_k|m_n, \boldsymbol{\eta}) = \frac{\alpha_k p(m_n|G'_k)}{\sum_{i=1}^K \alpha_i p(m_n|G'_i)}.$$

Summing up the responsibilities of one component grammar G'_k gives the effective number of models $N_k = \sum_{n=1}^N w_{kn}$ assigned to that component. The component weights of the mixture are then given by:

$$\alpha_k^{new} = \frac{N_k}{N}.$$

To update the rule probabilities in each component grammar, we use a modified version of the parameter learning approach explained earlier on. When adapting G'_k , instead of just counting the occurrences of rules in each model m_n , we weigh each occurrence with the responsibility w_{kn} .

To initialize the EM algorithm, we first cluster the training models in \mathbf{T} . Clustering is done by computing the pair-wise distances between all training models with a distance based on the kernel function. A neighborhood graph is created that contains edges between models if their distance is below a threshold (0.001 of the maximal distance of any two models). We iteratively remove the node with the highest degree and all its neighbors

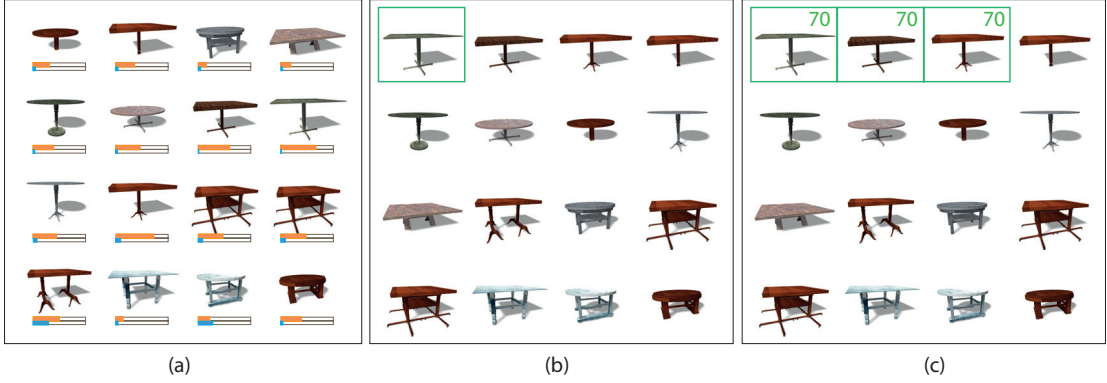


Figure 4.7: User Interface Users are presented with a grid of sample models from the grammar that they can score. There are various options to choose from for the distribution and sorting of the shown samples. (a) The sample models from the current pdf are displayed together with the prediction of the preference scores (orange) and the prediction uncertainties (blue). (b) The Models are sorted by similarity to the highlighted model. (c) Several samples are selected and scored simultaneously.

from the graph to create a new cluster. Per cluster, we initialize the hyperparameters for the component grammar with the normal parameter learning algorithm.

4.6 User Interface

Our user interface allows users to provide feedback by assigning preference scores to sets of sample models from a shape space. These scores range of 0 to 100 (where 100 is the most desirable) and our framework uses them to learn the preference function for the entire shape space using GPR (Sec. 4.4.2). We use an active learning approach that updates the preference function iteratively whenever a new set of scores is available, and the system will present the user with new samples in return. Fig. 4.7 shows screenshots of the user interface that arranges the sample models in a grid. The predicted preference scores and the prediction uncertainty can also be shown for each sample.

We support different settings for how the displayed sample models are chosen: *default*, *current pdf*, *complimentary pdf*, *uncertainty*, and *already scored*. *Default* samples the models from the default pdf. *Current pdf* samples from the pdf that is proportional to the current preference function. The *complimentary pdf* is a normalized version of 100 minus the current preference function. This will provide samples that the framework believes to be undesirable. *Uncertainty* shows models with high prediction uncertainty. This is useful because the framework gains more knowledge from scores for models that

	N_{Θ}	$N_{NT \cup T}$	N_P
Tables	50	38	75
Buildings	26	80	122
High-rises	39	27	61
Airplanes	8	44	26
Trees	73	N/A	N/A

Table 4.1: *Parameter Statistic* The table lists the number of parameters N_{Θ} , symbols $N_{NT \cup T}$, and production rules N_P for the four grammars and the Weber & Penn tree model.

the Gaussian Process is uncertain about. Finally, *already scored* shows all previously rated models and can be used to correct mistakes.

The order of the models in the grid can be sorted by preference score, by uncertainty, by similarity to a user-selected model, or it can be random. More complex preference functions are often easier modeled as a combination of factors (Sec. 4.4.3) that separately specify preferences for specific aspects such as color or shape. With the user interface, one can create, store, and combine factors.

4.7 Evaluation and Results

To evaluate our method we created four grammars for tables, buildings, high-rises, and airplanes. We further also use the Arbaro¹ implementation of the parametric tree model by Weber & Penn [WP95]. In Tab. 4.1 we give an insight of the complexity of the examples by listing the number of parameters, symbols, and rules.

In Tab. 4.2 we define several different design tasks for the grammars, each one with different complexity. For every task, we also define the ground truth as a set of up to 5000 manually scored sample models. We use Jensen-Shannon (JS) divergence [Lin91], a method for measuring the dissimilarity between two pdfs, to validate our results. We compare the pdfs learned with our framework with the target pdf of the ground truth. Figs. 4.8 (high-rises), 4.9 (airplanes), 4.10 (tables and buildings), and 4.14 (trees) show samples from the designed pdfs for all design tasks. Fig. 4.10 also presents examples that combine two design tasks in one. Combining the pdfs of the individual tasks as factors (Sec. 4.4.3) does this efficiently. For example, Fig. 4.11 shows that the pdf converges

¹Arbaro: <http://arbaro.sourceforge.net>, accessed on 2017/02/28

Preference Scores	
T1	Valid tables (that stand on their own and that have matching legs and bases) with one leg (70), two legs (20), four legs (10), and invalid tables (0).
T2	Bright wood tables with round top (60), dark wood tables with rectangular top (40), and all other tables (0).
T3	Valid tables made of steel (70) or wood (30), and invalid tables (0).
T4	Valid tables with round top (70) or rectangular top (30), and invalid tables (0).
B1	Buildings of different style: office buildings with glass windows, glass doors, and a flat roof (40); residential buildings with bright walls, Paris-style windows, and ground floor shops (20); residential buildings with simple windows and doors (20); residential buildings with old-school windows and doors (20); buildings with components with mismatching style (0).
B2	Big buildings (5-6 stories) with L-shape (50), small buildings (2-3 floors) with rectangular shape (50), and other buildings (0).
B3	Buildings of different shape: L-shape (60), rectangular (40), and other shapes (0).
B4	Buildings of different size: big (80), small (20), and other buildings (0).
H1	High-rises with all blocks being rectangular (60), V-shaped (20), or cylindrical (20). Other high-rises are (0).
H2	High-rises with rectangular (100), cylindrical (50), or V-shaped (0) bottom block.
A1	Old-school airplanes (e.g., double decker) (100), modern airplanes (commercial, transport and fighter jet) (50), and airplanes with mismatching parts (0).
WP1	Realistic trees (100) and implausible trees (0).

Table 4.2: *Design Tasks* The various design tasks for tables (T_i), buildings (B_i), high-rises (H_i), airplanes ($A1$), and Weber & Penn trees ($WP1$).

faster for the combination of tasks T3 and T4 when learned as two factors instead of one overall pdf.

We empirically find the parameters for the kernel and the ARD energy function, i.e., the noise-like term $\beta^{-1} = 0.01$ and the weight of the lasso regularizer $\lambda = 0.1$ (see Sec. 4.4.2). The publication corresponding to this chapter [DLC*15] and its supplementary material contain further evaluations and details that are unimportant for the contributions of this dissertation, e.g., a preliminary user study and measurements of the pdf convergence rate for the different user interface and framework settings. The supplementary material also contains more example models for each design tasks than shown in this section.

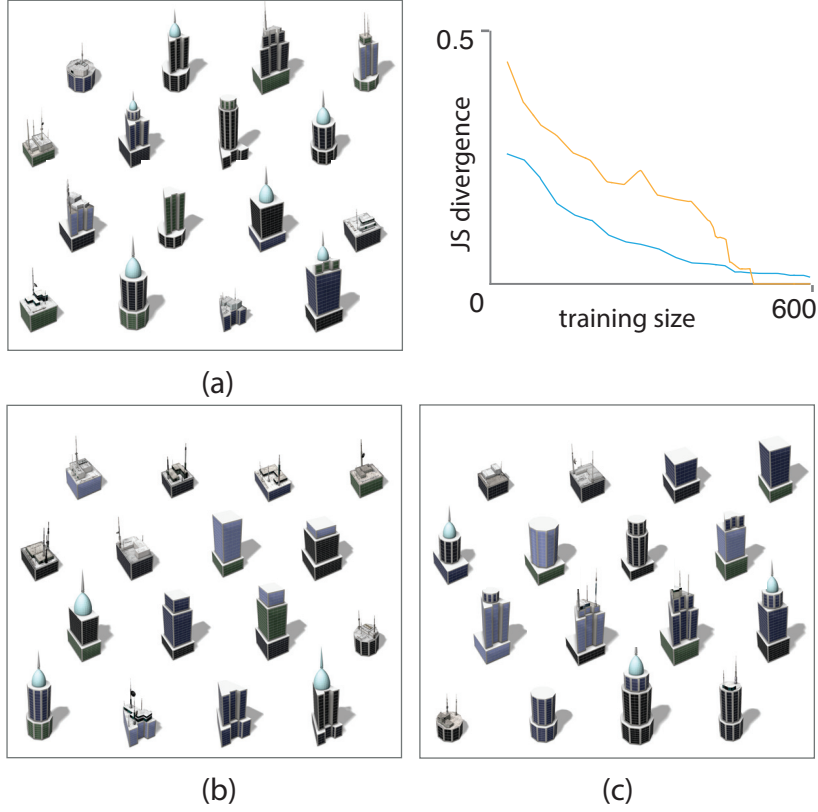


Figure 4.8: *High-Rise Buildings* Models from the grammar for high-rise buildings sampled from different pdfs: (a) default pdf, (b) pdf of design task H1, and (c) pdf of design task H2. H1 is learned with the initial features that contain only paths of effective length $k = 1$. An increase to $k = 2$ is needed for H2. A plot of the JS divergence for varying training set sizes shows the convergence rate and the quality of the approximation of the pdfs for both tasks (H1 in blue and H2 in orange).

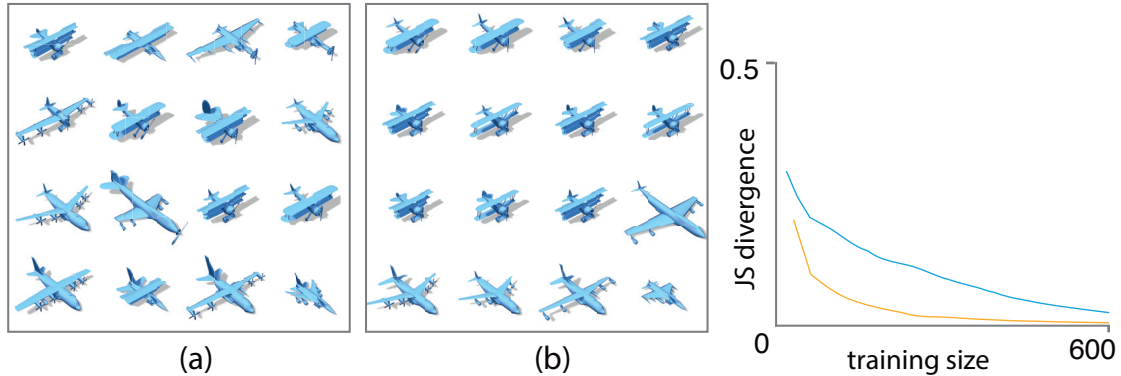


Figure 4.9: *Airplanes* Models of the airplane grammar sampled from (a) the default pdf and (b) the pdf of design task A1. We use JS divergence to compare the pdf convergence of our method (orange) with the method of Talton et al. [TGY*09] (blue).



Figure 4.10: Table and Building Design Tasks For the table and building grammars, we display a selection of models from the default pdf and sets of models sampled from the resulting pdfs for all the design tasks listed in Tab. 4.2. For several pairs of tasks we show that their preference functions can be combined as factors to define more complex preference functions. We plot the JS divergence for different numbers of training models for T1, B1 in orange, for T2, B2 in brown, for T3, B3 in red, and for T4, B4 in green.

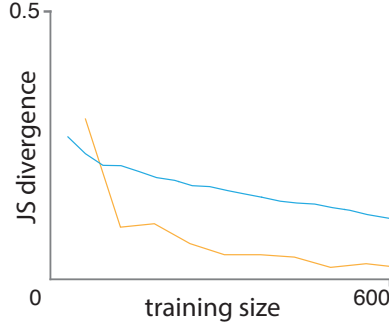


Figure 4.11: Convergence for Factors It is advantageous to model more complex preference functions as several factors. The JS divergences show that the combination of design tasks $T3$ and $T4$ can be learned more efficiently as two separate factors (orange) than as one overall pdf (blue).

Feature Evaluation Fig. 4.12 analyzes the effect of the effective path length k (Sec. 4.4.1) on the convergence behavior of our framework. We evaluate the JS divergence for the table and high-rise design tasks $T4$ and $H2$ for different amounts of training models. We disable the automatic adjustment of k and run the GPR for four different sets of features for maximal effective lengths 1, 2, 3, and 4. k does not have any influence for task $T4$. In task $H2$ however, we see that $k = 1$ (red) is insufficient to learn the pdf. For longer effective path lengths the target pdf is approximated well. But if it gets too large, e.g., $k = 4$ (black), the convergence is slowed down. From this, we deduce our strategy to start with $k = 1$ and to dynamically increase it if necessary.

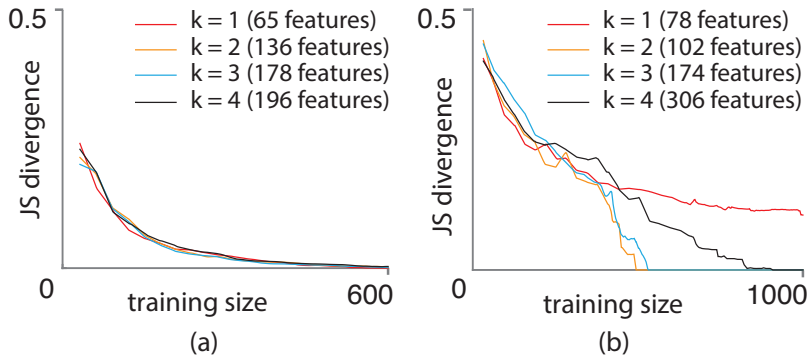


Figure 4.12: Evaluation of Effective Path Length The plots compare the JS divergence of pdfs learned with our framework with features sets with different maximal effective path length. We examine the table task $T4$ (a) and the high-rise task $H2$ (b).

4.7. Evaluation and Results

	T1	T2	T3	T4	H1	H2
Original	0.3282	0.4449	0.2804	0.2897	0.2465	0.3134
Parameter learning	0.2198	0.2539	0.1691	0.1854	0.0885	0.1963
Structure learning	0.0685	0.0668	0.0782	0.0840	0.0009	0.0816
	B1	B2	B3	B4	A1	
Original	0.3492	0.3483	0.0986	0.1197	0.2104	
Parameter learning	0.1054	0.2813	0.0071	0.1218	0.1675	
Structure learning	0.0666	0.0825	0.0071	0.0777	0.0627	

Table 4.3: Evaluation of Sampling Strategies The JS divergence with respect to the target pdf of the different design tasks in Tab. 4.2 is listed for the default pdf, the pdf obtained from parameter learning, and the pdf achieved by structure learning. The result of parameter learning is often only slightly better than the default pdf. Structure learning, however, achieves a good approximation of the target pdf.

Evaluation of Generation Strategies To analyze the effectiveness of the parameter and structure learning strategies (Sec. 4.5), we compare how well they can approximate the target pdf for each design task. For both strategies, we adapt the grammars based on ground truth models. The JS divergences of the pdfs of the adapted grammars and the target pdfs are listed in Tab. 4.3. For comparison we also provide the JS divergence of the default pdfs. The grammars adapted by parameter learning do not correlate well with their corresponding target pdfs. The only exception is the very simple design task B3, for which it is sufficient to change the rule probabilities of one single non-terminal symbol. Structure learning performs equally well for B3 and much better for all other design tasks.

Comparison to Kernel Density Estimation Our GPR-based framework is compared with the kernel density estimation method by Talton et al. [TGY*09]. We change Talton’s method to consider preference scores by using the scores as weights of the kernels. The JS divergence plots for selected tasks in Fig. 4.9 (A1) and Fig. 4.13 (T1, B1, H1) show that our method approximates the target pdf better for a given number of training models. For tasks B1 and H1 the pdfs learned with kernel density estimation do not correlate well with the target pdf even for a large set of training models. In Fig. 4.14 we also show the JS divergence comparison for the parametric Weber & Penn trees. Other advantages of our framework are that we can estimate the uncertainty of preference predictions, and we can also inform our system about unwanted models (by assigning preferences scores of 0) while Talton et al.’s method can only learn from desirable sample models.

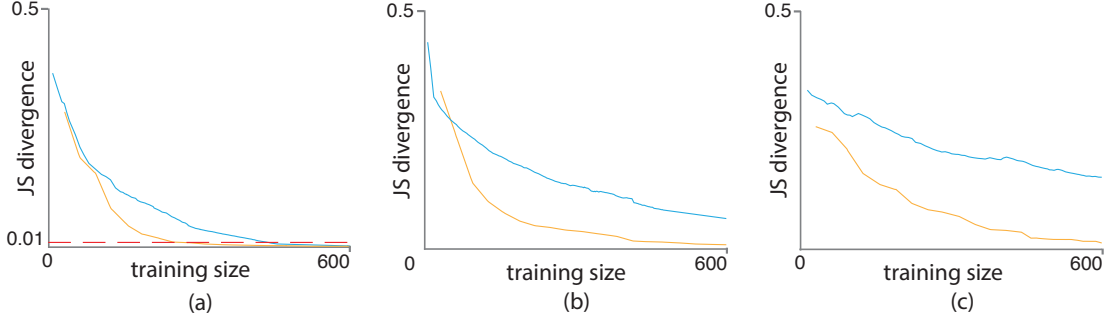


Figure 4.13: Comparison with Kernel Density Estimation The pdfs learned with our GPR-based method (orange) converge faster towards the target pdf than Talton et al.’s kernel density estimation method [TGY*09] (blue). The JS divergence is shown for table task T1 (a), building task B1 (b), and high-rise task H1 (c). For task T1, our method reaches a JS divergence of 0.01 after scoring 245 sample models while Talton’s method needs 438 samples.

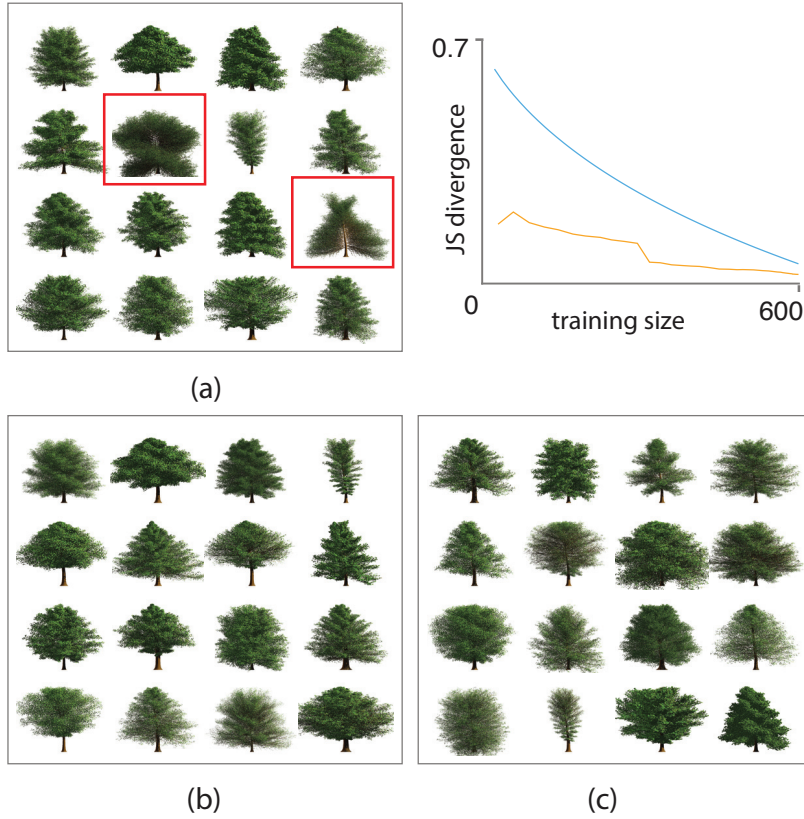


Figure 4.14: Weber & Penn Trees (a) Random parameter combinations can lead to unrealistic tree models like the two highlighted examples. Talton et al.’s method [TGY*09] (b) and ours (c) can both adapt the parametric model’s distribution to avoid such undesirable trees. The JS divergence of both methods shows that our method (orange) converges faster than Talton et al.’s kernel density estimation (blue).

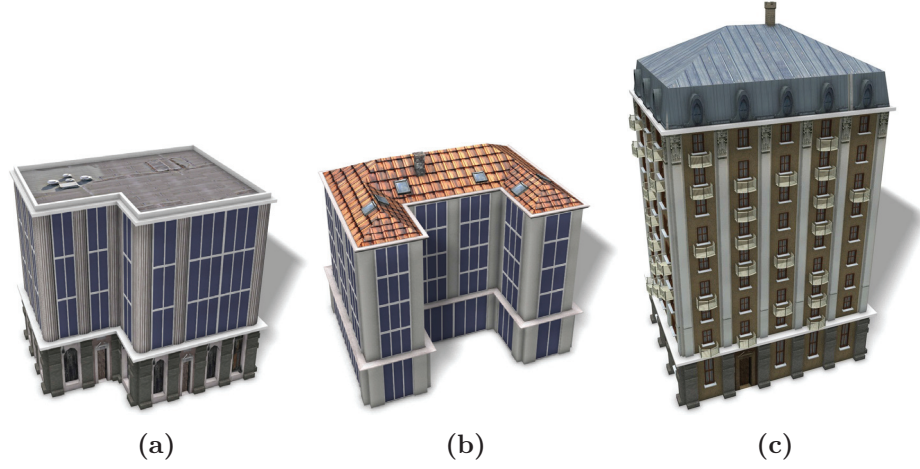


Figure 4.15: *Style Mismatches* Some of the visible style mismatches of building components in Fig. 4.16 (top) are: (a) Mismatch of ground and upper floor styles: a house with an ancient ground floor should not have its higher floors in a modern office style with glass façades. (b) Mismatching roof and façade: modern office buildings do not usually have old red tile roofs. (c) Too many floors: the quintessential Parisian-like Haussmannian building should not be larger than six floors.

4.7.1 Urban Planning Use Case

Consider the task of modeling a city using an existing shape grammar that can generate a variety of different buildings. If the grammar is general enough to cover a broad range of styles over multiple building components such as roofs, façades, windows, etc., then the initial result could look very chaotic, e.g., as in Fig. 4.16 (top). The buildings have random materials and a random height distribution. Such a grammar can also be a combination of different more specific grammars that were merged together.

Some of the generated models also do not make sense because they contain mismatching styles and colors within the model. In Fig. 4.15 we illustrate several such example mismatches from the chaotic city that can occur when the grammar is too general.

While it is possible for an expert to manually edit the grammar to enforce all required design constraints, it is not a scalable solution. As more components from different architectural styles are added to the grammar, it becomes more complex to encode all their interdependencies, and maintenance of the grammar will become a tedious task due to the combinatorial explosion. With our framework even novice users can model pdfs for the building grammar that avoid these mismatches.

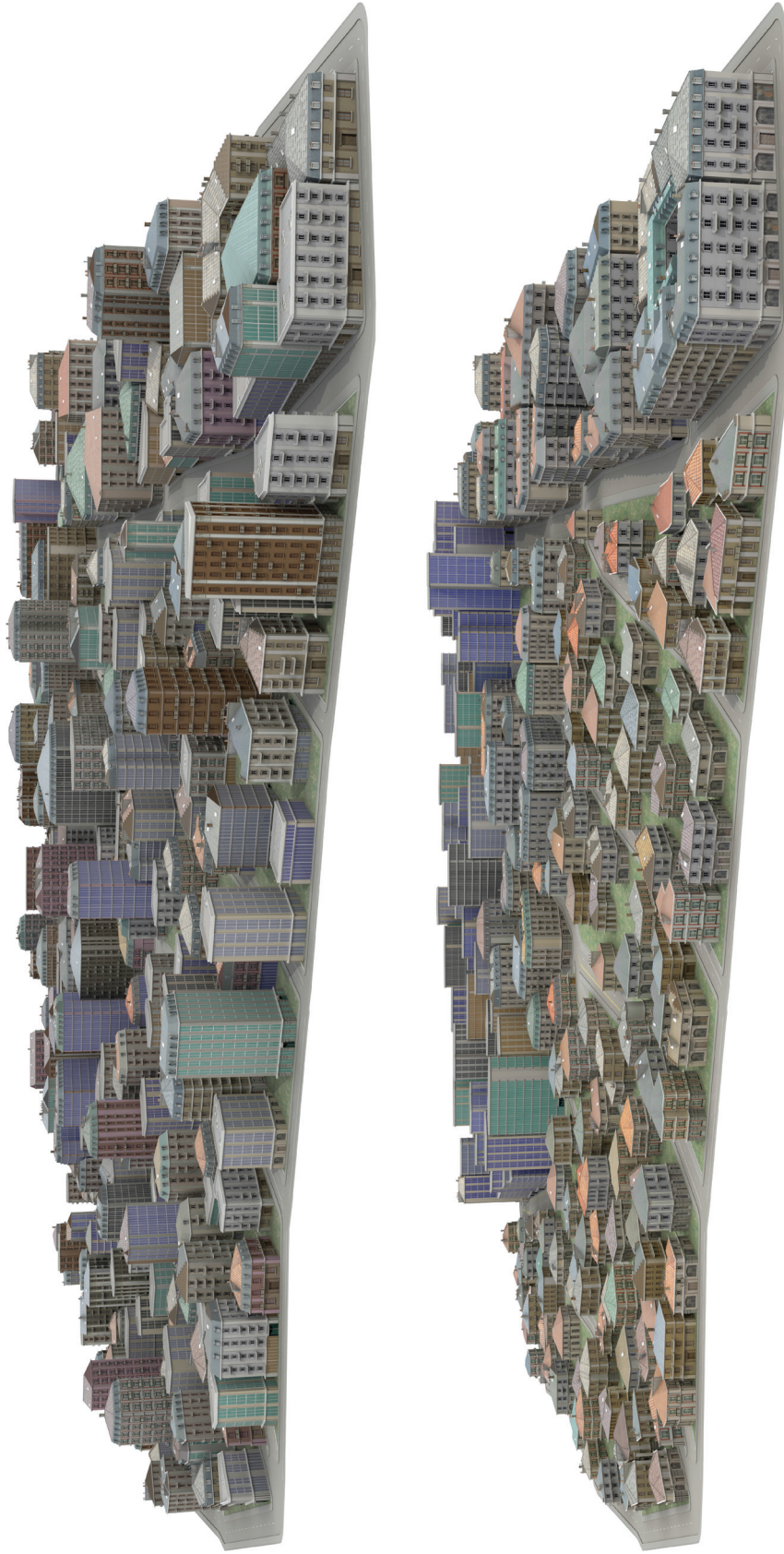


Figure 4.16: Urban Planning Use Case We use a grammar that generates a variety of buildings for a city modeling task. (top) A direct application of the grammar leads to undesirable results. For example, office buildings are mixed with Haussmannian buildings and small residential houses without a clear structure of different neighborhoods. (bottom) We use our framework to design three different probability density functions for this grammar, which bias the generation towards high-rise office buildings (far), downtown Haussmannian buildings (right) and residential houses (left). We can also ensure the matching of house styles, roofs, and wall colors.

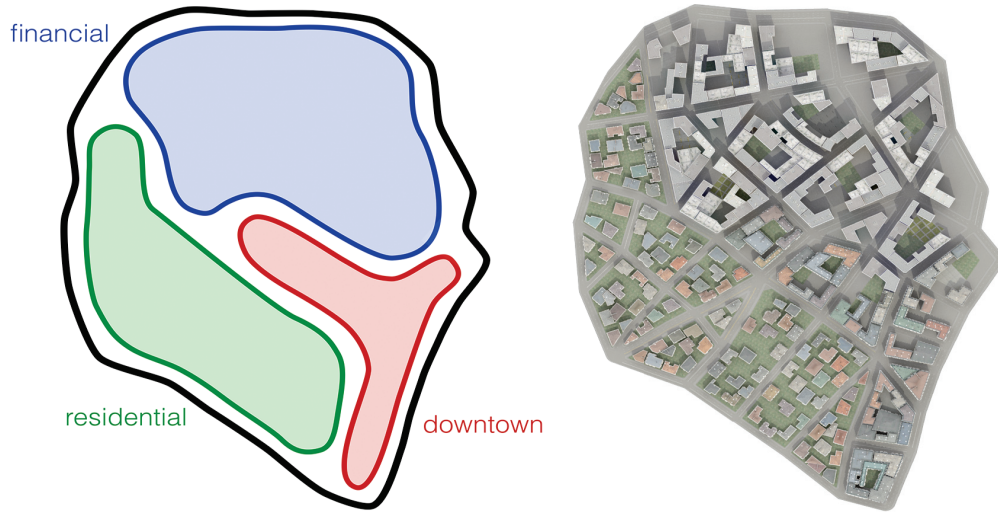


Figure 4.17: Top View of City Districts The left shows the layouts of the residential, downtown, and financial districts. On the right is a rendering of the final city.

Imagine that we want to divide our example city into three distinct districts as shown in the top-down perspective in Fig. 4.17: a residential region for small suburban buildings, a financial district with office buildings, and a downtown area featuring Haussmannian architecture. To achieve our design objective, we specify two goals: 1) each district should contain buildings appropriate for that district (e.g., a skyscraper would look strange in the residential area, and a residential building would be out of place amongst the glass buildings in the financial district), and 2) no mismatches in any buildings. With our system the user can easily achieve these goals by designing three individual pdfs, one for each district. Each pdf limits the grammar’s shape space to the subset of valid models for the given district. Our resulting city in Fig. 4.16 (bottom) was created from a single grammar by sampling from these three pdfs to generate three different sets of buildings: short residential buildings, Haussmannian downtown buildings, and office buildings.

4.8 Limitations and Future Work

Limitations Our work has some limitations. A common problem of GPR is its inability to learn discontinuities. This is problematic when there is a discontinuity in a preference function that depends on a continuous variable. An infinite number of training models would be required to learn the exact function because it cannot be represented with a finite number of Gaussians. For our framework, this problem can be overcome when such continuous variables are changed in the grammar to discrete sets of values. Further,

the more features are involved the slower the pdf converges to the target pdf (e.g., in Fig. 4.12 (b)). For very complicated design tasks requiring feature sets with long paths, it is unavoidable that a large number of sample models are scored. Also, the complexity of possible design tasks is limited by the design of our features. Paths in the shape tree can only follow one child branch at every node they pass through. Hence it is impossible to encode design constraints that depend on different child branches of the same node. For example, in the high-rise grammar that recursively stacks building blocks on top of each other, our feature vector cannot encode that cylindrical blocks with black windows are preferably on top of box blocks with green windows but not on top of box blocks with blue windows. The properties of the different levels are in different branches of the shape tree. It would take subtrees and not just paths to capture this aspect. *Tree kernels* [CD01] used in natural language processing (NLP) do exactly that. Instead of counting paths, a tree kernel counts the occurrences of all possible subtrees in a parse tree. While tree kernels inspired us, the same methodology is not applicable in our case, simply because our examples have much larger shape/parse trees than the common examples in NLP. The number of features in a tree kernel increases exponentially when the tree grows. Finally, our framework always begins the learning process with the default pdf from the input grammar. If the user is interested in models that are unlikely in the initial distribution, it can take many iterations before the framework displays any desirable samples.

Future Work It would be interesting to extend the framework to also learn the spatial distribution of models, e.g., the distribution of different tree species in a forest. In the urban planning use case (Sec. 4.7.1), it could help to create smooth transitions between the residential, downtown, and commercial districts. Another future project might take the user out of the loop. Any black box system that can evaluate the goodness of a certain type of model could be used to assign scores. For example, a physical simulation could assess the structural stability of sample models and rate them accordingly. An adapted grammar could avoid instable models. A different direction for future research could also improve the design the features.

4.9 Conclusions

Our framework allows users to design pdfs for stochastic shape grammars and to adapt the grammar so that it creates more models according to the designed pdf. Both expert

and non-expert users profit from our system to encode difficult design constraints in such procedural models as grammars and parametric models. Our contributions extend the domain of exploratory modeling from finding single models to designing probability distributions of entire shape spaces. We propose a new kernel function that compares different derivations of a given grammar and use it in GPR to interpolate user-assigned preference scores from a few sample models over the rest of the shape space. Non-terminal split operations increase the degrees of freedom of stochastic grammars without altering their shape space. This allows adapting grammars to derive models according to the user-designed target pdf.

5 Design Transformations for Rule-based Procedural Modeling

We introduce *design transformations* for rule-based procedural models, e.g., for buildings and plants. Given two or more procedural designs, each specified by a grammar, a design transformation combines elements of the existing designs to generate new designs. We introduce two technical components to enable design transformations. First, we extend the concept of discrete rule switching to rule merging, leading to a very large shape space for combining procedural models. Second, we propose an algorithm to jointly derive two or more grammars, called *grammar co-derivation*. We demonstrate two applications of our work: we show that our framework leads to a larger variety of models than previous work, and we show fine-grained transformation sequences between two procedural models.



Figure 5.1: Tree L-Systems A design transformation computed from two different L-systems (first and last images). Our method can compute a sequence of intermediate results transforming one design into the other by combining and merging the L-systems rather than the geometry of the two trees.

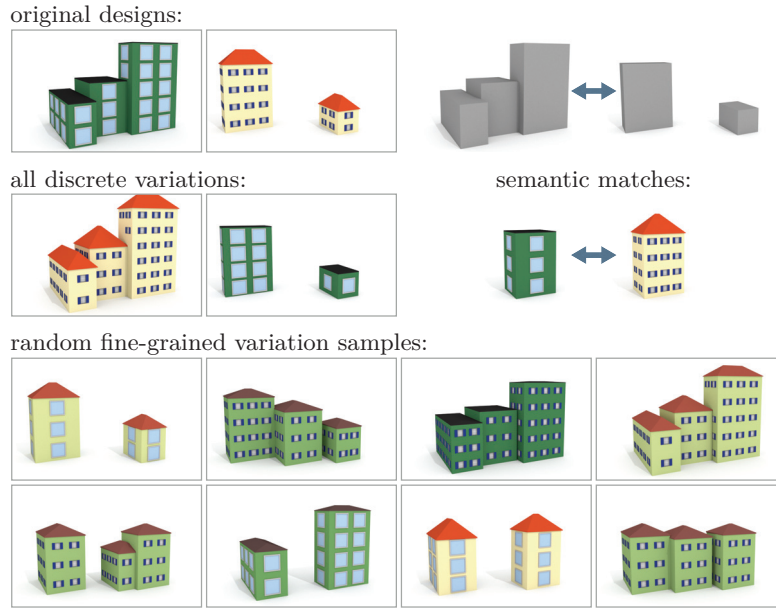


Figure 5.2: Coarse-Grained Versus Fine-Grained Transformations Two simple grammars have correspondences (called *semantic matches*) for the production rules that generate the mass models and the building style (top right). Discrete rule switching spans only a very limited shape space with four designs, the two original designs (top left) plus two variations (middle left). Fine-grained rule merging can span a shape space with infinitely many designs (bottom).

5.1 Introduction

Architectural design over centuries gives us many examples for the reuse, evolution, and the combination of designs [SMR04, Kni94]. We study these *design transformations* and present algorithms for their technical realization in the context of rule-based procedural modeling.

Rule-based procedural modeling systems, such as L-systems and shape grammars, provide an efficient method for the automatic creation of scenes with rich geometric detail. The rules specify how to iteratively evolve a design from a crude initial shape to a highly detailed model.

The simplest way to realize design transformations is to switch rules between different procedural models, e.g., to replace the window rule of one building with the window rule of another building. *Rule switching* can be used to combine two or more procedural models using the framework of Bayesian model merging [TYK*12].

discrete transformation sequence:



different fine-grained transformation sequences:

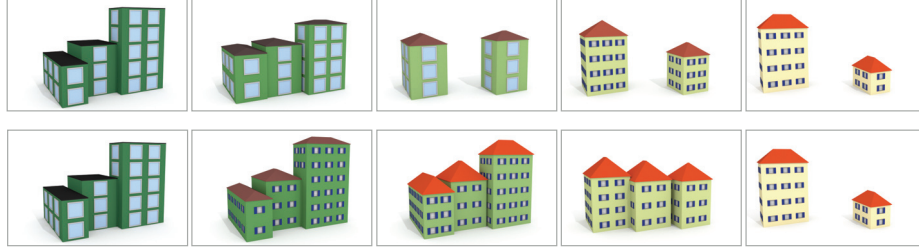


Figure 5.3: Transformation Sequences Discrete switches can only create very limited transformation sequences while fine-grained rule merging enables the generation of different interpolation paths that morph one design into another.

We improve on this initial idea. Rule switching is the simplest method to obtain design transformations, but it only provides coarse-grained control and a limited number of new designs. In our framework we complement rule switching with *rule merging*, i.e., combining the effect of two rules. In contrast to rule switching, rule merging also provides continuous changes and generally leads to an infinite number of variations. For our solution we need to overcome two major challenges. First, we need to be able to merge rules that are structurally different, e.g., they generate a different number of shapes and place them in different spatial arrangements. Second, grammars also exhibit structural differences globally. As a consequence, it is often not possible to establish simple one to one correspondences between shape (rule) labels. In such cases we can only establish sparse correspondences and it is not sufficient to merge individual rules. We propose a solution to this problem by merging shape trees that are the result of (partial) derivations using multiple rules. We call this new derivation framework *grammar co-derivation*.

To summarize, we present the following main contributions. On the technical side, we propose algorithms for rule merging for rule-based procedural modeling. On the application side, we can generate a larger space of design variations than competing methods (Fig. 5.2) and demonstrate fine-grained transformation sequences between different procedural designs (Figs. 5.1 and 5.3). Transformation sequences are gradual transformations of one design into another, and they can be shown as computer animations. The generation of such transformation sequences is difficult to achieve using existing work.

5.2 Related Work

An inspiration for this project was Terry Knight’s book *Transformations in Design* [Kni94]. It describes how certain architectural styles evolved over time. The book goes further than just defining the styles with shape grammars; it introduces *meta rules* that do not operate on the shapes but rather modify the grammar’s production rules. The evolution of a style can thus be represented by a sequence of meta rules that are applied to the shape grammar. Knight’s work is based on the archetypal shape grammars by Stiny [Sti82]. While she discusses architectural concepts, our work focuses on the technical aspect of automatically computing design transformations. We demonstrate our idea of co-derivation both in the context of grammars similar to CGA shape [MWH*06] and L-systems [Pru86].

In this section we only mention literature that is relevant to our design transformations. For a complete overview of rule-based procedural modeling techniques, please refer to Chap. 2.

Inverse Procedural Modeling and Grammar Induction In computer graphics there are several recent approaches that study how to generate design variations given a single existing design. Bokeloh et al. [BWS10], Stava et al. [ŠBM*10], and Talton et al. [TYK*12] all propose methods to compute simple shape grammars from input designs to generate variations of those input models. The main difference between our work and previous work is the granularity of the combination. While previous work mainly relies on rule switching, our grammar co-derivation and rule merging operations lead to a much larger shape space.

Morphing and Style Transfer Our design transformations combine elements of volume morphing and style transfer — two successful modeling methods in computer graphics. Volume morphing was pioneered by Leros et al. [LGL95] and Kanai et al. [KSK97]. An overview over different morphing methods is given by Alexa [Ale02]. Similar to our work are the methods by Alhashim et al. [ALX*14, AXZ*15] that do not only blend shapes geometrically but also topologically.

Style transfer is the process of applying the style of one exemplar model to another target model. Often geometric high frequency details are re-targeted and applied to another mesh [BIT04, MHS*14]. The same idea has also been applied to images [HJO*01].

Structure Preserving Editing There are structure-aware methods that automatically keep the finer geometric details intact while the coarse shape of a 3D model is changed [CLDD09, BWKS11, BWSK12]. Our method also has to deal with geometry changes at the structural and detail levels. We have the advantage to work with rule-based procedural models; if a design changes at a coarse level, the structure of the detail shapes can be kept consistent by reevaluating their production rules. Our challenge is to keep that consistency when combining elements from different grammars.

Design Exploration The space of all possible design transformations for a set of grammars could be interpreted as a parametric model that spans a shape space. That space could be navigated with an interactive exploration tool, e.g., the work by Talton et al. [TGY*09] or our own method from the previous chapter [DLC*15].

5.3 Overview

5.3.1 Grammar Definitions

As our proposed concepts are applicable to a broader range of rule-based procedural modeling systems, we describe our framework for a generalized set grammar, similar to CGA shape [MWH*06] and L-systems [Pru86]. We define a grammar G as a four-tuple:

$$G = \langle NT, T, \omega, P \rangle.$$

The grammar operates on shapes where each shape is assigned exactly one label, or *symbol*. The symbols stem from two disjoint sets: non-terminals (NT) and terminals (T). A shape can have a list of geometric and non-geometric attributes. The most important attributes are encoded by the *scope*, i.e., a local coordinate frame and the shape’s size defining a (bounding) box in space. The grammar derivation starts with a single shape labeled with the special symbol $\omega \in NT$, the *axiom* or *start symbol*. During the grammar derivation, a shape is selected and a *production rule* (or just *rule*), is applied to it. P is the set of rules of the form:

$$predecessor : cond \rightarrow successor,$$

where *predecessor* is a symbol $\in NT$ and *successor* is a general procedure (or program) that generates zero, one, or more successor shapes labeled with symbols ($\in NT \cup T$).

A rule can only be applied if its condition *cond* is met. A condition can depend on shape attributes (e.g., the scope’s size, position, etc.), the rule’s parameters (in case of parametric grammars), and other shapes. In our examples, we use conditions for occlusion queries [MWH*06] in building grammars and as a recursion counter in plant grammars (the max number of derivation steps is initialized to n and decreased during derivation so that a rule can stop the recursion when $n < 0$). We use the term *deterministic* grammar if there is only a single possible derivation for all possible starting shapes. Otherwise, a grammar is called *stochastic*. Procedural modeling systems differ in the way a *successor* can be defined. Typically, commands like translation, scaling, rotation, instantiation of mesh and texture assets, and splitting rules are used.

The derivation generates a *shape tree*. After applying a rule to a non-terminal shape, all the shapes generated by *successor* will be added as children. We assume the derivation order to be breadth-first.

5.3.2 Framework Overview

Our framework consists of multiple components. In a preprocess we establish sparse correspondences between rules of the input grammars (Sec. 5.4.1). These correspondences help our proposed *co-derivation* (Sec. 5.4.2) to synchronize the simultaneous derivation of two grammars. The co-derivation algorithm uses two different strategies for rule merging: 1) a general shape matching and blending algorithm (Sec. 5.4.3) and 2) a specialized algorithm for split rules (Sec. 5.4.4). We further describe the extensions necessary for our two applications together with their results: co-derivation using multiple grammars is important for generating variations of designs (Sec. 5.5.1), and we introduce a user interface to control transformation sequences (Sec. 5.5.2).

5.4 Co-Derivation of Shape Grammars

5.4.1 Sparse Correspondences

We require a sparse set of correspondences between the symbols (rules) of the input grammars. These correspondences are called *semantic matches*. In general, this is a modeling problem that does not have a single correct solution, and we let the users adjust the semantic matches according to their design intent. We use the tuple notation

$\langle Sym_a, Sym_b \rangle$ to say that symbol Sym_a in one grammar matches symbol Sym_b in another.

Since the structure of the grammars can be quite different, not every rule will have a valid match. For the presented examples, only 24-54% of the rules have matches. We allow one-to-many matches, i.e., it is possible that a rule has more than one match in another grammar. For example, one grammar might have different rules for *Gothic Window* and *RoundWindow* which both match the only *Window* rule in the other grammar. The axioms of all grammars are always set to be a semantic match.

Finding semantic matches automatically is a very challenging problem and beyond the scope of this dissertation. We have the user annotate selected rules with tags coming from a set of architectural vocabulary (e.g., mass model, roof, façade, floor, window tile, window, etc.). Rules with the same tag are automatically matched while non-tagged rules are not matched. This assignment works in most cases. Otherwise the user can manually refine the semantic matches.

5.4.2 Grammar Co-Derivation

Grammar co-derivation is an extension of traditional grammar derivation to two grammars. We first look at the problem of extending a single derivation step, and then use our proposed solution to design a complete grammar co-derivation algorithm.

The elementary step of a traditional shape grammar is to select a shape, select a rule, and replace the shape by its successor by adding newly generated shapes as children in the shape tree. We call the elementary step of grammar co-derivation, used to derive shapes with semantic matches, a *co-derivation step*. The co-derivation step generates separate partial derivations of a shape x for each grammar and merges both outputs. The fundamental challenge of designing a co-derivation step is the following: if we only apply a single rule from each of the two input grammars to x , we likely end up with two sets of incompatible shapes. Some of these will have semantic matches and others will not. However, good merging results can only be achieved if we work with two sets of shapes where all of them either have semantic matches or are terminal shapes. A co-derivation step therefore needs to synchronize the derivation. Each grammar needs to derive shape x until only terminal shapes or shapes with semantic matches exist. Our co-derivation step has three parts (also see Fig. 5.4):

1. Derive x with both grammars. Similar to traditional grammar derivation, we also use breadth-first order, but stop at shapes that have a semantic match or that are terminals. See Alg. 5.1.
2. Collapse the two subtrees. For each of the two subtrees, we bring all shapes into a common coordinate system and delete intermediate nodes.
3. Merge all shapes of both collapsed subtrees into a final shape set. The merging can either combine shapes from the two derivations, discard shapes, or copy shapes unmodified. The resulting shapes are appended to the shape tree as x 's children. The merging operation is more involved and we will explain two merging algorithms in Secs. 5.4.3 and 5.4.4.

To define a complete co-derivation, we recursively apply co-derivation steps until all the shape tree leaves become terminal shapes. See Alg. 5.2 for pseudo code.

Algorithm 5.1 `coDerivationStep_1(shape, grammar)`

```

1: derive(shape, grammar)
2: repeat
3:   done = true
4:   for each  $l \in \text{leaves}(\text{shapeTree})$  do
5:     sym = symbol( $l$ )
6:
7:     # keep deriving non-terminal leaves without semantic matches
8:     if  $\nexists$  semantic match for sym && sym  $\in NT$  then
9:       derive( $l$ , grammar)
10:    done = false
11: until done

```

5.4.3 Rule Merging

The input to this step are two sets of shapes, one from grammar G_a and one from grammar G_b . The output of this step is a single set of shapes that are either directly copied from G_a or G_b and/or shapes that are combined by blending pairs of shapes. In general, this is an underspecified problem with many possible solutions. In order to manage this large design space, we propose the following method to parameterize the solutions. For each semantic match, we use a parameter $p_s \in [0, 1]$. $p_s = 0$ means that all shapes are directly copied from grammar A while $p_s = 1$ means all shapes are directly copied from grammar B. In addition, we use a single threshold th (default 0.5) to facilitate certain discrete decisions, e.g., when to switch assets.

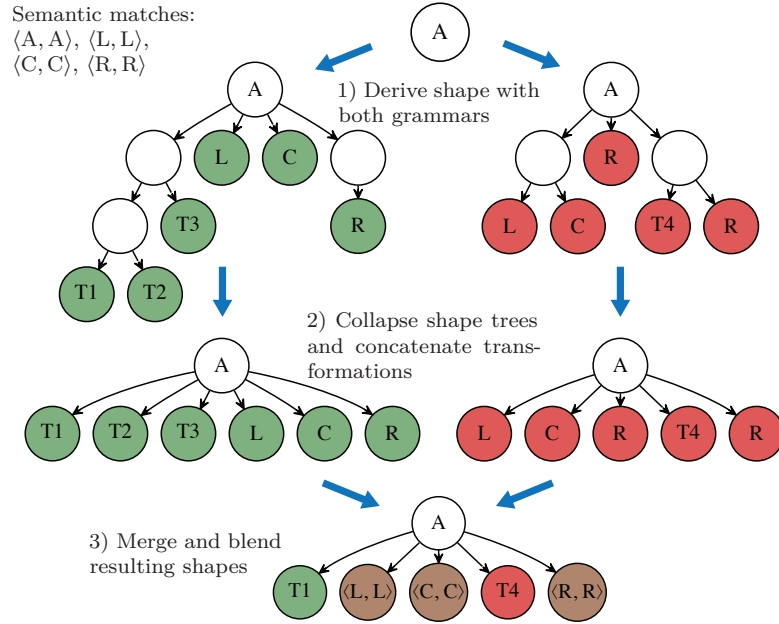


Figure 5.4: The Co-Derivation Step 1) A shape, whose symbol A has a semantic match, is separately derived with both grammars. 2) Both subtrees are collapsed and all leaf shapes are transformed into a common coordinate system. 3) The resulting shapes are combined in a merge step according to user preferences. Shapes can come from the first (green) or the second (red) grammar. If a shape's symbol has a semantic match, it can also be blended together with matching shapes from the other subtree (brown). Blended shapes are labeled with the match tuple.

Algorithm 5.2 Co-Derivation Process

```

1:  $shapeTree = \text{new node}(\langle Axiom, Axiom \rangle)$ 
2:
3: repeat
4:   for each  $l \in \text{leaves}(shape\_tree)$  do
5:     if  $\text{symbol}(l) \notin T$  then
6:       # derive twice and collapse shape trees to height 1
7:        $st_1 = \text{copy}(l)$ 
8:        $\text{coDerivationStep}(st_1, \text{grammar}_1)$ 
9:        $\text{collapse}(st_1)$ 
10:       $st_2 = \text{copy}(l)$ 
11:       $\text{coDerivationStep}(st_2, \text{grammar}_2)$ 
12:       $\text{collapse}(st_2)$ 
13:
14:    # calculate resulting set of shapes
15:     $res = \text{mergeAndBlend}(\text{leaves}(st_1), \text{leaves}(st_2))$ 
16:     $l.append(res)$ 
17: until  $shape\_tree$  does not grow anymore
    
```

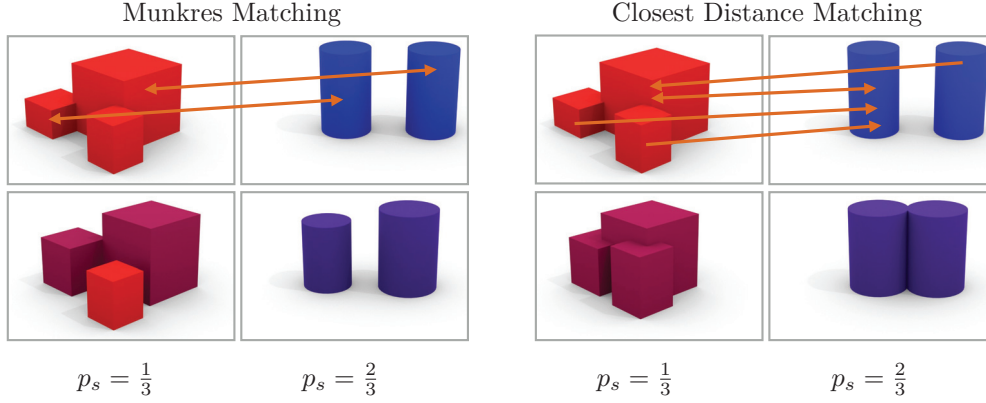


Figure 5.5: Shape Matching & Blending The two automatic strategies for shape matching are Munkres assignment (top left) or closest distance matching (top right). Munkres matches are always bidirectional. (bottom) The merged and blended resulting shape sets are shown for the parameter values of $\frac{1}{3}$ and $\frac{2}{3}$ with a threshold of 0.5.

To compute the resulting set of shapes, we need three ingredients: 1) a distance metric that can be computed between each pair of shapes, 2) a matching algorithm that computes matches based on the distance metric, and 3) an interpolation algorithm that combines matched (and unmatched) shapes.

We propose a distance metric d that is the sum of two components, a semantic distance d_s and a geometric distance d_g . The semantic distance d_s is zero if there is a semantic match between the labels of the two shapes and infinity otherwise. That means that two shapes can only be matched if there also exists a semantic match between their corresponding symbols. The distance-metric d_g compares shape positions and scope sizes by summing up the point-to-point distances of the eight scope corners.

We propose two automatic ways to compute matches based on the metric d . (We allow the user to override the automatic assignment for additional artistic control, but this option is not used in any of our results.) 1) The first strategy uses the Munkres (or Hungarian) algorithm [Mun57] for finding a minimum weight maximum matching in a weighted bipartite graph. It results in only one-to-one and one-to-none matches. 2) The second matching algorithm assigns each shape from G_a 's output the closest shape derived by G_b and vice versa. Note that this assignment, unlike the result of the Munkres algorithm, is not necessarily symmetric. All pairs of shapes that mutually map to each other are included in the set of pairs. The Munkres matching method might leave remaining single (unmatched) shapes while the closest distance approach will likely end up with some unidirectional matching pairs.

Our proposed interpolation algorithm works as follows. Singles from G_a and unidirectional pairs pointing from G_a to G_b are kept if the corresponding parameter p_s is below the threshold th and vice versa. The advantage of this approach is that it also allows one-to-many matches. A shape matching and blending example is given for both proposed strategies in Fig. 5.5. The shapes of matching pairs are blended by interpolating their scope attributes according to the corresponding parameter p_s . Position, size, and color are linearly interpolated, while we use quaternion *slerp* to interpolate the shapes' orientations with respect to their centers. For overlapping blended shapes, an additional parameter decides if they are kept as is or if they are merged together (which enables interesting scenarios such as the one with multiple houses merging into one building shown in Fig. 5.12). Currently, we do not support the morphing of asset geometry attached to the shapes. We simply use the threshold to switch between the assets from G_a and G_b . The same applies to all textures. In the output, all blended shapes are labeled with tuple names.

5.4.4 Rule Merging for Split Rules

Split rules [WWSR03, MWH*06] are a particular type of rule to define the *successor* of a shape. They are most commonly used for architectural modeling. A split rule subdivides a shape along an axis into a set of smaller shapes that are tightly aligned. For example, a split rule along the y-axis can partition a façade into individual floors. An advantage of split rules is that they can be written to adapt to the size of shapes (encoded by the scope). For example, a split rule can generate a reasonable number of floors for façades of any height. An important aspect of split rules is that they partition the scope, i.e., none of the successor shapes should overlap, and all the space defined by the scope should be filled by successor shapes. The previously described solution cannot guarantee these two conditions, because the shapes produced by a split rule cannot be transformed independent of each other. For example, a shape in the split pattern can only become larger if other shapes shrink. Our method for handling split rules guarantees that the output is also a valid split pattern. We again use a parameter for semantic matches to parametrize the solution space.

Our proposed algorithm has two steps, structure computation and geometry computation. The first step computes a sequence of symbols (thereby deciding on the number, type, and relative position of symbols). The second step computes the shape attributes, such as size.

Structure Computation For example, a string for a split pattern of a building floor consisting of windows (W), pillars (P), and wall pieces (ω) could be:

$$\{\omega, W, \omega, P, \omega, W, \omega, P, \omega, W, \omega, P, \omega, W, \omega, P, \omega, W, \omega\}.$$

In building designs the split patterns consist of relevant shapes such as ornaments or windows. For our split transformation algorithm that is tailored towards such architectural patterns, the wall pieces that make up the spacing between the important elements are irrelevant. The user indicates the symbol that is used for spacing (default *Wall*). The example pattern is reduced to:

$$\{W, P, W, P, W, P, W, P, W\}.$$

The removed wall shapes will be reinserted again in the geometry computation step.

We propose an algorithm based on an edit distance between two strings. We optimize the edit distance and take the lowest cost edit sequence to compute intermediate strings. The algorithm takes two reduced strings as input (*src* and *dst*) and produces a sequence of intermediate strings. A single parameter value p_r will define what string of the sequence is used at the current state. The edit distance is computed by summing the cost of elementary edit operations:

- *delete* – Removes a single symbol from the string. Only applicable if the symbol (or a semantic match) appears in *src* and *dst* but more often in *src*.
- *insert* – Inserts a single symbol. Can be used if the symbol (or a semantic match) appears in *src* and *dst*, but more often in *dst*.
- *switch* – Exchanges the positions of two different, neighboring symbols.
- *delete all* – Removes all symbols with a given label that only appears in *src*.
- *insert all* – Inserts all symbols with a given label that only appears in *dst*.
- *semantic switch* – Replaces all symbols of a label with semantically matching symbols.

Since split rules are predominantly used for regular patterns with repeating elements or with a clear structure, we also want edit operations that keep such regularities intact. For strings that are symmetric, i.e., they read the same from the left and the right (like a palindrome), we define symmetric versions of the delete, insert, and switch operations. These symmetrically apply the same edit operation twice. Further, if a string is symmetric,

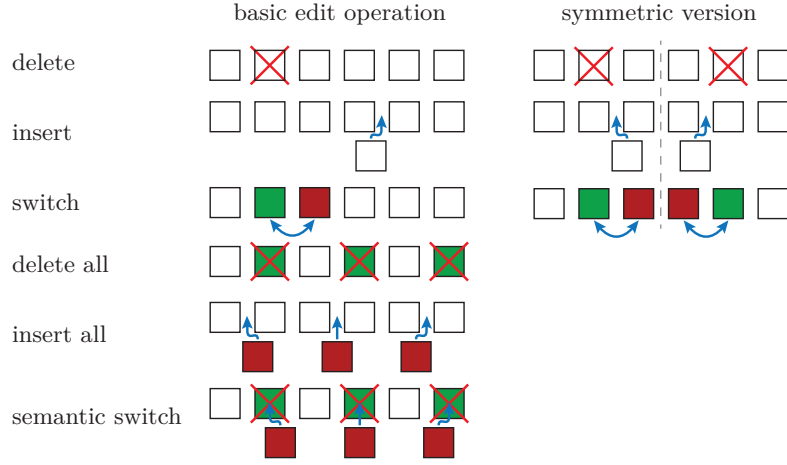


Figure 5.6: Edit Operations Overview of all edit operations for split transformations. For details see Sec. 5.4.4.

edits that result in a symmetric string again will be preferred over edits that break the symmetry. All edit operations are summarized in Fig. 5.6. Two simple examples of what is possible with such edits are shown in Fig. 5.7.

Each possible edit operation has an assigned cost that reflects how drastic the change of the string is. The task of finding a smooth sequence of strings that gradually changes from *src* to *dst* can be cast as a shortest path problem on a graph that can be solved with Dijkstra’s algorithm. Graph nodes represent strings while each edit operation and its cost correspond to a weighted edge. We never store the entire graph in memory since we do not know it upfront. We expand it on the fly; when the search arrives at a specific string, we grow the graph by adding all remaining neighbors of the current node. All edit operations except the symmetric ones have a cost of 1. If an edit breaks the string’s symmetry its cost is increased by 100 to assure that a non-symmetry breaking sequence

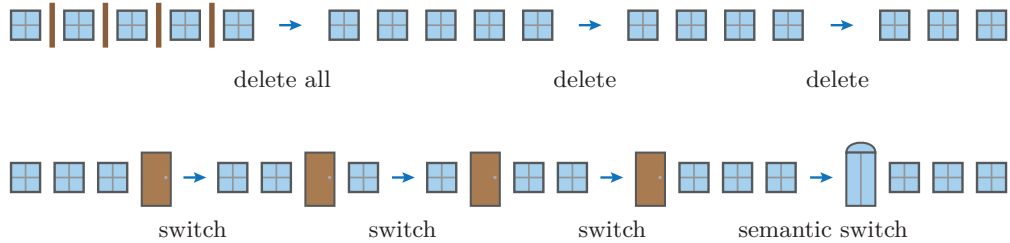


Figure 5.7: Edit Sequences Two simple sequences that apply edit operations. (top) First all pillars are removed, then the number of windows is gradually reduced to three. (bottom) A door is moved from the left to the right with subsequent switches, then a semantic switch changes the door style.

is given priority (if one exists). Symmetric edit operations cost 2.5. They are less gradual than two normal edit operations because they change two symbols at once. However, it is often not possible to apply two subsequent normal edit operations without breaking the symmetry. To prevent the search from exploding, we also penalize edits that result in strings that are longer or shorter than the two input strings. This speeds up the search without changing its outcome.

Geometry Computation This step of the algorithm sets the shape sizes and adds spacing between the elements by reinserting walls. Our observation to make this algorithm work is that we not only need to consider the sizes of shapes in the two input sets but also the average sizes of shapes occurring in the model in general. We fully derive all input grammars and compute the average sizes of all shapes with a specific symbol and the average distances between neighboring shapes. If a symbol in the output has a semantic match, the average shape sizes from grammars G_a and G_b are interpolated according to the parameter p_r . The average distances are used to reinsert *Wall* shapes of that size. If two symbols do not appear as neighbors in the input, the average spacing between one symbol and any other symbol is used instead. The sum of the sizes of all resulting shapes might not exactly match the size of the parent scope. Therefore we uniformly rescale the *Wall* shapes to match the size. In Fig. 5.8 we show an example result of our method for transforming split rules applied to two hierarchical patterns consisting of several nested splits.

5.5 Applications & Results

We describe how we use and adapt the basic design transformation framework to enable our two modeling applications, and we present their results.

5.5.1 Variety Generation with Multiple Grammars

To compute a co-derivation of multiple input grammars, we extend our framework. When working with n grammars, a semantic match tuple can have a maximum size of n . To derive shapes with tuple labels, we randomly pick two of the participating grammars that define a rule for that symbol and apply the co-derivation step described in Sec. 5.4.2. To generate a random variety of shapes, we can randomly sample parameters for the

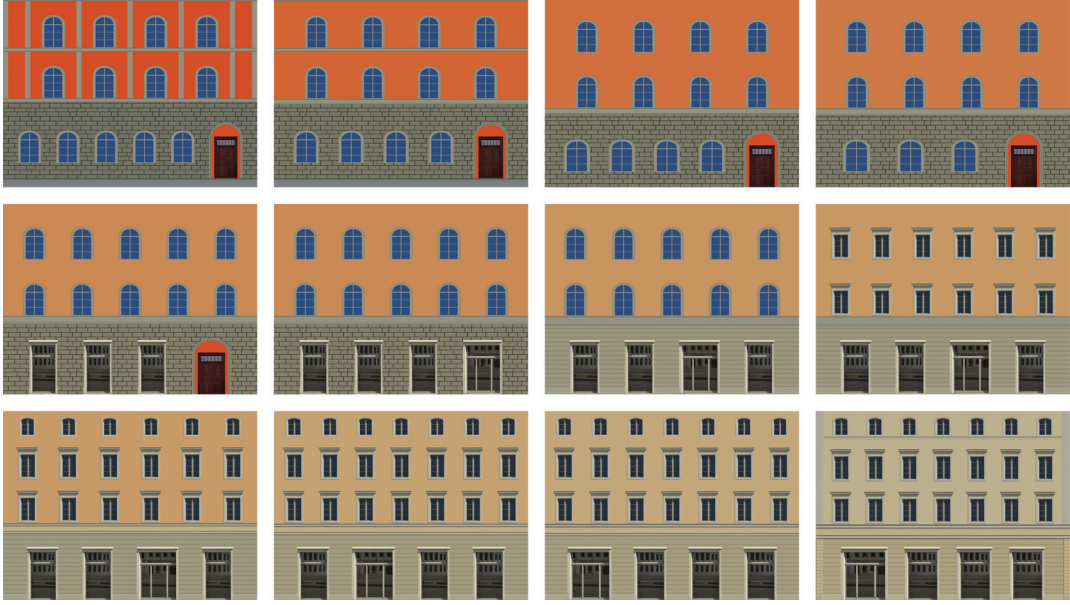


Figure 5.8: *Façade Split* A transformation between two hierarchical façade split patterns. The first (top left) and last (bottom right) images show the two inputs to the algorithm, and the remaining eight images show intermediate steps. The entire transformation applies 18 gradual changes to façade structure over the timeline. For elements with semantic matches, scope sizes and shape colors are smoothly interpolated.

semantic matches and their thresholds during the derivation.

We explore the concept of generating a variety of designs in the context of urban modeling. The idea behind our approach is that it is much easier to write deterministic grammars depicting a few designs (for example by recreating designs from photographs) rather than writing a stochastic grammar describing a larger shape space. Design transformations can then be used to generate a city consisting of new unique models of the same architectural style. The input to our example are four deterministic exemplar grammars depicting Venice style buildings. The output is a city consisting of about 400 buildings (Fig. 5.9). The buildings are the result of randomly combining and transforming parts of the different Venice grammars.

To compute the design transformations we use the shape blending algorithm from Sec. 5.4.3 for the block and roof rules and the split rule merging algorithm from Sec. 5.4.4 for the façade rules.

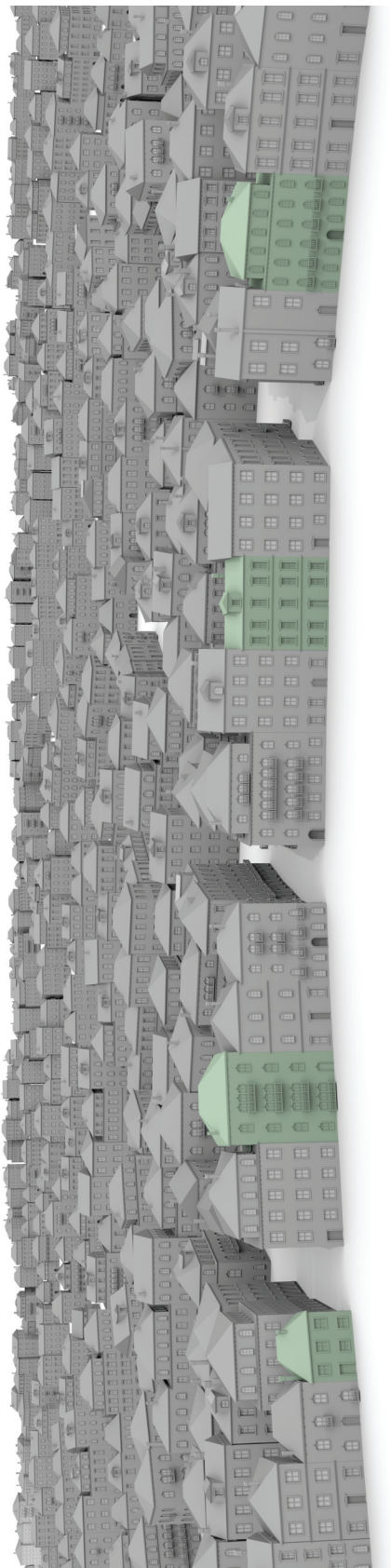


Figure 5.9: Venice Variations Design transformations enable the generation of new variations from a small set of exemplars. The four original Venice buildings are highlighted in green. Random design transformations combine and blend elements from the different exemplars to generate arbitrary new building variations to populate the rest of a virtual Venice.

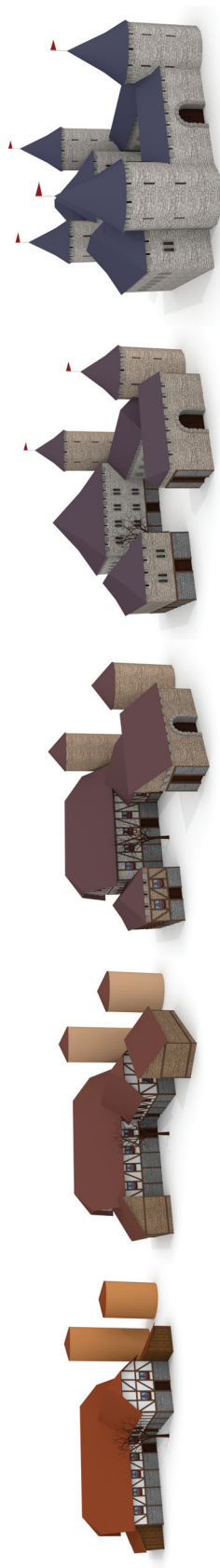


Figure 5.10: Farm \rightarrow Castle A farm gradually transforms into a castle. The farm consists of six mass model components (house parts, sheds, and silos) and the castle of eight (towers, walls, and main halls).

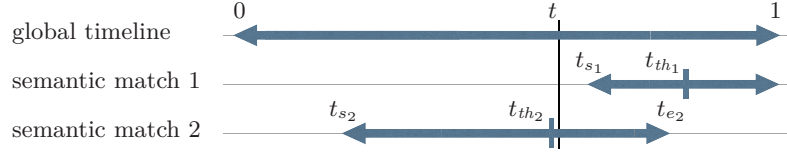


Figure 5.11: User Interface A sketch of the user interface for making transformation sequences. The transformation of the first semantic match is gradually executed over the time range $[t_{s1}, 1]$ and has not started at t . The transformation for the second semantic match is active at t since $t \in [t_{s2}, t_{e2}]$. t_{th1} and t_{th2} represent the thresholds that control discrete decisions. For semantic match 2, asset geometry and textures are taken from the second grammar since $t > t_{th2}$.

5.5.2 Transformation Sequences

One application of our work is to compute a transformation sequence between the design of one grammar and the design of another. There are two conflicting goals for designing transformation sequences. First, all intermediate designs should be valid. Second, the transitions between intermediate designs should be as smooth as possible. For example, a valid building design needs to have windows of a certain minimum size, which requires discontinuities in the transformation sequence. However, these should be limited as much as possible to favor smoothness. Similar animations are often used as special effects in the entertainment and movie industry. We control transformation sequences by a global timeline parameter $t \in [0, 1]$. All individual transformation parameters for each semantic match are derived from the global timeline. The user has the possibility to change the default mapping by adjusting start and end times and thresholds. By default all parameters span the full range and all thresholds are set to 0.5. Fig. 5.11 illustrates the user interface.

We show transformation sequences for building and plant models. All sequences can have alternative interpolations paths. See Fig. 5.3 and the Sternwarte example described in the following.

Farm \rightarrow Castle In Fig. 5.10 a farm transforms into a castle. Both consist of several mass models that are matched as follows: *Shed* and *MainBuilding* symbols match *MainHall*, *CastleWall*, and *Gate* symbols, and *Silo* corresponds to *Tower*.

Residential Houses \rightarrow High-Rise Fig. 5.12 shows the transition of a group of several small residential houses into fewer but larger apartment blocks. From there the scene



Figure 5.12: *Houses* \rightarrow *High-Rise* A transformation from several small residential houses into apartment blocks that eventually converge into a high-rise office building. For this scene we allow one-to-many shape matches, and we decide to collapse intersecting mass model shapes.

transforms into a high-rise building. Merging the buildings together in such a fashion is facilitated by one-to-many matches.

Chain of Transformations We use four procedural buildings to create transformations between them: the Sternwarte building designed by Semper, two modern residential buildings, and a Hausmannian building from Paris. Transforming the buildings in the given order leads to three transformation sequences that can be concatenated together as shown in Fig. 5.13. An alternative transformation of the first part, in which all façade elements change before the mass models, is shown in Fig. 5.14. In App. A.1 we give detailed descriptions of the parameters and settings of both versions of the first part of this transformation chain.

This scenario was created as an homage to Müller et al.’s video¹ [MWHG05] in which Semper’s Sternwarte model is morphed into Corbusier’s famous Villa Savoye. With our system it is easier to create such animations because it avoids laborious manual keyframing tasks.

Tree L-Systems The first plant transformation sequence (Fig. 5.1) is based on two procedural models presented by Honda [Hon71] and by Aono and Kunii [AK84]. The example illustrates how a monopodial branching pattern smoothly changes into a sympodial one. In both cases, an apex will always create exactly two offspring branches. An in-depth description of this tree L-system transformation is presented in App. A.2.

¹<https://www.youtube.com/watch?v=deSpf9EGcvw>, accessed on 2017/02/28



Figure 5.13: *Sternwarte Chain* An animation sequence that consists of several sequential transformations. Semper's *Sternwarte* design is first changed into a white modern building, then into a red modern building, and finally into a Hausmannian building.

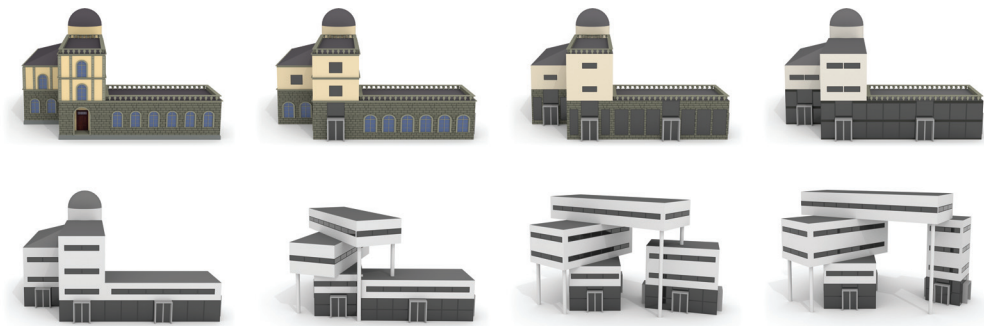


Figure 5.14: *Sternwarte Chain 1 Alternative* An alternative interpolation path for the same transformation as shown in the first row of Fig. 5.13. This time all façade rules are transformed before any of the mass model rules.



Figure 5.15: Flower L-Systems A more complicated transformation sequence that gradually changes the output of one flower L-system into another. Design transformations do not only blend the overall tree structure but also the smaller leaf and blossom shapes. What further complicates the transformation is that the growing rules for both flowers are structurally very different.

Flower L-Systems A more complex plant example (Fig. 5.15) is inspired by the L-systems in Figs. 1.26 and 3.14 in Prusinkiewicz and Lindenmayer’s book [PL90]. Not only does this example feature leaves and blossoms whose colors, shapes, and sizes are interpolated, but the branching structure is also more complicated. In each iteration, the first grammar grows each apex into two subsequent internodes and three new apices, one between the internodes and two at the end of the outer internode. The other grammar grows an apex into an internode followed by one younger and one older apex. At the next iteration the older apex recursively repeats the branching, while the younger apex grows into an older one (which will only expand later). The rules have to be carefully matched so that the merge and blend steps happen after every iteration for the first L-system but only after every other iteration for the second L-system.

5.5.3 Quantitative Results

Our implementation of design transformations builds on a newly designed procedural modeling system that combines elements of CGA Shape, G^2 [KPK10], and L-systems. We implemented the framework in C++ and used a 2012 MacBookPro to measure the running times. See Tab. 5.1 for the timings of our algorithm and other quantitative information of the examples in this chapter.

Name	Figs.	Time [s]	#Terminals $\times 10^3$	#Rules	#Matches	%NT w. Match	#Params	#Edits
Tree L-Systems	5.1	0.11	0.5	23	6	39	10	0
Fine-grained Variations	5.2, 5.3	0.05	0.6	34	4	24	10	0/6
Façade Split	5.8	0.02	0.2	51	15	54	30	0
Venice Variations	5.9	0.06	0.8	108	15	46	25	0
Farm \rightarrow Castle	5.10	0.18	3.3	117	20	24	50	22
Houses \rightarrow High-rise Part 1	5.12	0.20	1.8	48	9	38	14	3
Houses \rightarrow High-rise Part 2	5.12	0.23	2.9	49	9	37	14	4
Sternwarte Chain Part 1	5.13, 5.14	0.15	1.0	107	21	36	39	17/23
Sternwarte Chain Part 2	5.13	0.06	0.7	60	12	40	22	4
Sternwarte Chain Part 3	5.13	0.10	1.5	85	13	27	23	7
Flower L-Systems	5.15	0.04	0.3	42	8	39	18	4

Table 5.1: Results We list the following results (and their corresponding figures): the average computation time per transformed model (mean over 200 frames or variations), the average number of terminals shapes in the design, the number of rules of all involved grammars, the number of semantic matches, the average percentage of rules with matches (%NT w. Match), the number of parameters, and the number of parameters adjusted by the user (#Edits). If a result has two different interpolation paths, we show two different numbers of user edits. The percentage of rules with matches is an indicator of the sparsity of the correspondences. It cannot simply be inferred from the number of rules and semantic matches since certain rules are part of more than one match. The number of parameters hints at the large size of the shape spaces. Comparatively few user edits are usually sufficient to get a reasonable result.

5.6 Discussion

Our two applications clearly show how design transformations facilitate the creative process of combining and merging parts of different grammars. It is important to note that our framework does not need stochastic grammars as input but works directly with deterministic grammars. While we can also apply our method to stochastic grammars, modeling deterministic grammars is much easier for the user. Therefore, our framework can help a user to accelerate modeling larger environments because the time-consuming modeling step of extending deterministic grammars to stochastic ones can be omitted.

Difference to L-Systems Our design transformations are not directly applied to L-systems. Instead we recreate the procedural plant descriptions using our grammar framework. The main reason is because traditional L-systems derive a complete model as a string in a first pass and then interpret the string as a geometric model in a second

pass. In contrast, our grammar interpreter provides a geometric interpretation at every intermediate step, as required by our transformation framework. Plant models also require two minor changes in the derivation. First, rotations are not interpolated around shape centers, but are interpolated around the attachment points at the scope origins. Second, the output of a merging operation needs to be a valid branching structure. This can be achieved by enforcing connectivity as a constraint in the merging step.

Comparison with Talton et al. [TYK*12] Their method for Bayesian grammar induction has similarities with our first application insofar that it also allows to generate variations from a small set of input models. For Talton et al.’s method, the inputs are given as scene graphs annotated with semantic labels. These trees are automatically converted into trivial, deterministic grammars. By applying ideas from natural language processing, a more general and stochastic grammar can be inferred. This works very well for the examples presented in their paper. All inputs and also the resulting grammars are, however, Lego-like: all designs are composites of predefined components with given connector locations (similar to Plex languages [Fed71]). Our approach is more flexible. It can also deal with more advanced operations such as extrusions, component and subdivision splits, etc., and not only stick existing pieces together.

Comparison with Knight [Kni94] With rule switching it is always possible to define a new (transformed) grammar that can derive a transformed design. For rule merging however, this is not the case anymore. Our rule merging solely operates on the shapes that are output by the production rules. In essence, we generate a valid shape tree by combining and mixing elements of several grammars. None of the involved grammars alone can generate such a shape tree on its own. This also clearly distinguishes our project from Knight’s more abstract work: she uses meta rules to describe how production rules of a grammar change while always having a valid grammar at each step. While our approach does not operate on the grammars themselves, it works with the output of set grammars that can handle more complex models than Knight’s schematic and purely geometry-based shape grammars.

5.7 Limitations and Future Work

Our work has several limitations and possibilities for improvements. First, it would be better if semantic matches could be found fully automatically without any manual annotations. This is a very hard problem since one single mismatch can impair or even completely ruin the result. It is elusive and some human knowledge or preference will always be required. Some initial results for a related problem were presented by Alhashim et al. [ALX*14, AXZ*15]. However, this problem becomes significantly more difficult for our inputs since semantic matches might exist between parts of the design that are geometrically and structurally different.

We aim for transformation sequences that are as smooth as possible, but there are still noticeable discrete changes when transforming one model into another. The reason is that many objects, such as windows, cannot realistically grow from zero size, but they need to start appearing with a realistic minimum size. This leads to some discontinuities in the transformation. Switching terminal assets also leads to small discontinuities; it would be interesting to morph them via convolutional Wasserstein barycenters [SdGP*15].

Our edit operations for split rules can transform many regularly structured splits, but not all of them. It would be helpful to expand the system with more edit operations, e.g., with one that removes one repetition out of a repeat split. Further research could also lead into the direction of transforming less regular or even random split patterns.

5.8 Conclusions

We present design transformations as a modeling tool for rule-based procedural modeling. On the application side, we show two improvements of the state of the art. First, we show how design transformations can create a larger set of variations from a set of input designs than previous work. Second, we show how to compute fine-grained transformation sequences between two input designs that cannot be generated with previous work. We describe two main components that enable design transformations: *grammar co-derivation* and *rule merging*. The results show example design transformations for a variety of building and plant models.

6 Conclusions

In conclusion, we present algorithms that address some of the drawbacks of current rule-based procedural modeling systems. More specifically, we contribute 1) a pipeline that visualizes the diversity of models in the shape space of a given grammar, 2) a framework that allows users to adapt the probability distribution of the models generated by a grammar according to their design intent, and 3) a method to transform between designs defined by different grammars. Each system is different, remedies a different limitation of rule-based procedural modeling, and makes shape grammars more accessible to novice users.

We introduce thumbnail galleries for rule-based procedural models to help users visualize and grasp the range of variations that grammars can define. This is helpful when one is confronted with several grammars to choose from. We show how a new set of view attributes allows us to find the best view of a procedural model and compare two different models derived from the same grammar. Based on these view attributes, the pipeline samples a grammar’s shape space, clusters the sample models, ranks them, and arranges selected representatives rendered from their best viewpoint in a resulting thumbnail gallery. Such galleries provide an intuitive visual understanding of a shape space.

We demonstrate that stochastic grammars with undesirable default distributions can be adapted to reflect a user’s intent. A grammar’s default distribution also often contains completely unwanted models that are avoided with our method. New distributions for different tasks are designed with an active learning approach that builds on Gaussian process regression. Grammar parameters and shape tree paths are used as features in a kernel function to compare models generated by the same grammar. The important

features vary for different design tasks, and they are detected by automatic relevance determination. We show how splitting relevant non-terminals provides a solution for the difficult problem of modifying a grammar to generate new models according to a designed distribution. Also novice users can use the system as no understanding of the grammar is required. Additionally, the framework can be used to explore the shape space of a grammar.

Finally, design transformations solve the non-trivial task of combining elements of different grammars to emerge new designs. Our framework is guided by the user and first creates semantic matches to establish correspondences between grammars. Then we apply grammar co-derivation to jointly derive several grammars rules. It provides fine-grained control for merging the outputs of the rules at different stages of the derivation process. Design transformations allow the generation of innumerable variations from a set of procedural designs by randomly combining and merging their different components. We can also create animated transformation sequences between two procedural designs.

6.1 Future Work

There are ideas for future research directions that are directly related to the three main topics of this dissertation, e.g., improving the sampling strategy for the thumbnail galleries, replacing the user with a black box when learning a pdf and improving the feature design, or finding semantic matches automatically. These have already been mentioned in the discussions of the corresponding chapters (Sec. 3.6, Sec. 4.8, and Sec. 5.6). It would further be interesting to see if the features used for designing distributions for shape spaces could also be used in the clustering step of the thumbnail gallery pipeline. We would expect these features to perform better because they were developed after the view attributes with the unique purpose of distinguishing models.

Although our three main contributions ameliorate some of the limitations of rule-based procedural modeling systems, there are still remaining problems. Some of the major advantages of most rule-based systems are also their drawbacks. The context-freeness and also the implicitly defined control flow of shape grammars keeps the syntax simple and makes shape grammars easy to use, but they unfortunately also hinder the expressiveness for modeling complicated structures that have to interact and query each other. We believe that research towards more expressive shape grammars without overcomplicated syntax will become important.

Ideally, in the future we could combine all these techniques, and more, in one common tool. For example, this tool would feature a grammar database together with an interactive user interface to browse and explore grammars. At first, thumbnail galleries would provide previews of grammars. The shape spaces of promising grammars could then be explored in more detail with our learning approach by iteratively designing distributions that generate models close to the current model(s) of interest. If several models are selected, design transformations would allow users to mix and merge their components (with an extension to stochastic grammars and an automatic way to find meaningful semantic matches). We also hope that novel machine learning techniques will give inverse procedural modeling techniques a boost. The grammars in the database should not be written manually anymore but rather learned directly from acquired data such as point clouds or photos. We envision a future for shape grammars that will provide a full tool set not only to visualize, adapt, and transform grammars, but also to acquire or learn them automatically from various sorts of input data.

A Detailed Design Transformation Result Descriptions

In this appendix we provide detailed descriptions of the grammars, the semantic matches, their parameters, and the entire co-derivation process for two of our results in Chap. 5. We first provide some insights into the transformations between Semper’s Sternwarte and the white modern building via two different interpolation paths (Figs. 5.13 and 5.14). Second, we explain how we transform the two tree L-systems in Fig. 5.1. We use the notation `@tag` to annotate rules that we consider for semantic matches (Sec. 5.4.1). For each possible pair of two rules (one from each grammar) with the same tag, a semantic match is automatically established.

A.1 Sternwarte Chain Part 1

Mass Models The first co-derivation step derives the start shape with the axiom rules of both grammars. It passes through some intermediate (untagged) rules before it comes to a halt after all mass model shapes have been instantiated. Their rules are tagged with `@mass_lvl0`, `@mass_lvl1`, or `@mass_lvl2` for shapes on the ground plane and shapes at two higher stacking levels. The tags essentially define a synchronization barrier that allows to apply a match & blend step (Sec. 5.4.3) to the mass models before the co-derivation continues. Fig. A.1 shows the result of the shape matching.

Afterwards, in the second co-derivation iteration, the rules split the mass models into their faces for façades and roof tops, and they also generate railings on the Sternwarte roofs and support columns for the white modern building. The component splits of both grammars always produce the same exact shapes. For them, the interpolation parameters have no influence on the outcome of the co-derivation steps since the matched

Appendix A. Detailed Design Transformation Result Descriptions

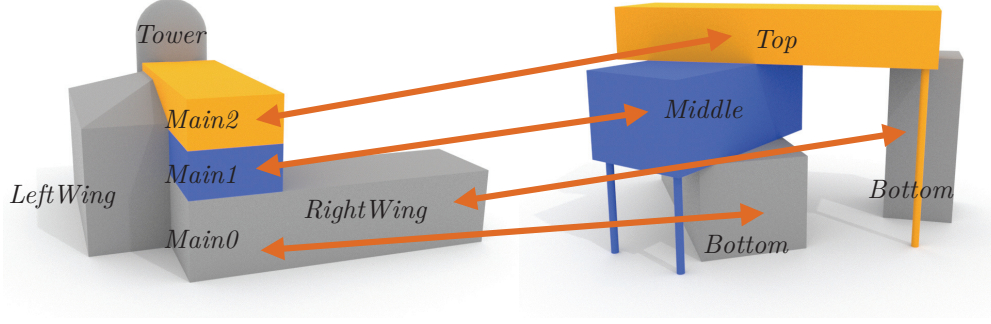


Figure A.1: Shape Matching The result of the Munkres matching for the first part of the Sternwarte Chain. Only shapes with the same colors can be matched. Gray shapes are tagged as `@mass_lvl0`, blue shapes as `@mass_lvl1`, and orange shapes as `@mass_lvl2`. The unmatched shapes `LeftWing` and `Tower` will disappear during the transformation while the other shapes are blended with their matches.

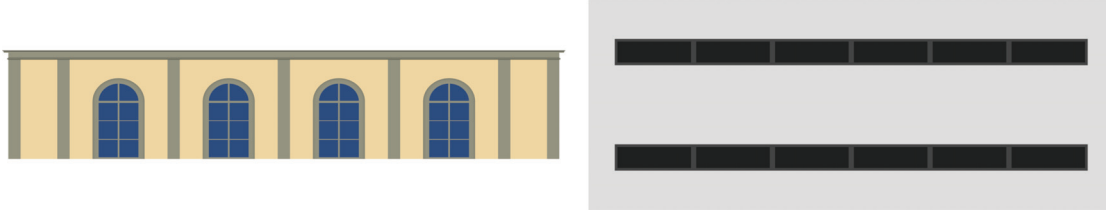


Figure A.2: Sternwarte and White Modern Building Façades The two façade styles for the upper mass models of both building designs.

and blended shapes are identical. But the railings and columns are only defined in one of the two grammars, and if these elements are present or not depends on the thresholds of the semantic matches that control the co-derivations ($\langle Main2, Top \rangle$ and $\langle RightWing, Bottom \rangle$). The thresholds are visible in Figs. A.3 and A.4.

Façade Transformations The next co-derivation steps of interest are the split transformations of the façades. We hierarchically apply the technique presented in Sec. 5.4.4. As example, we describe the transformation of the façades shown in Fig. A.2. They are the façades of the long sides of the blue shapes in Fig. A.1. The automatically computed edit sequence for that $\langle YellowFacade, Facade \rangle$ co-derivation is:

$\{ Floor_1, Ledge \}$	delete all <i>Ledge</i>
$\{ Floor_1 \}$	semantic switch $Floor_1 \rightarrow Floor_2$
$\{ Floor_2 \}$	insert <i>Floor₂</i>
$\{ Floor_2, Floor_2 \},$	

where we use the subscripts to distinguish symbols that occur in both grammars. The edit sequence for the next lower level, controlled by the semantic match $\langle \textit{Floor}, \textit{Floor} \rangle$, is given in the following. The notation is: P for *Pillar*, Yt for *YellowWinTile*, and Wt for *WindowTile*.

$\{ P, P, Yt, P, Yt, P, Yt, P, Yt, P, P \}$	delete all P
$\{ Yt, Yt, Yt, Yt \}$	semantic switch $Yt \rightarrow Wt$
$\{ Wt, Wt, Wt, Wt \}$	insert Wt
$\{ Wt, Wt, Wt, Wt, Wt \}$	insert Wt
$\{ Wt, Wt, Wt, Wt, Wt, Wt \}$	

Window tiles are transformed as vertical splits. Their strings only consist of one single *Window* shape (after removing the *Walls*) and the edit sequence is a single semantic switch. Even though the edit sequence is very simple, the split transformation still provides a nice effect because the size interpolation smoothly shrinks the window over time.

Parameters In Figs. A.3 and A.4 we list all semantic matches together with their parameters for both interpolation paths of the transformation between the Sternwarte and the white modern building. For some semantic matches, some of the parameters are unused or ignored and we do not show them in the figures. Possible reasons for that can be:

- If a shape is not the result of a blending operation and if the shape does not initiate a split transformation, t_s and t_e are not used.
- If no discrete decisions need to be made, t_{th} is not needed.
- Split transformations do not need thresholds either.
- Parameters are irrelevant when both grammars generate identical output in a co-derivation step.
- The parameters do not matter if a semantic match is never applied.

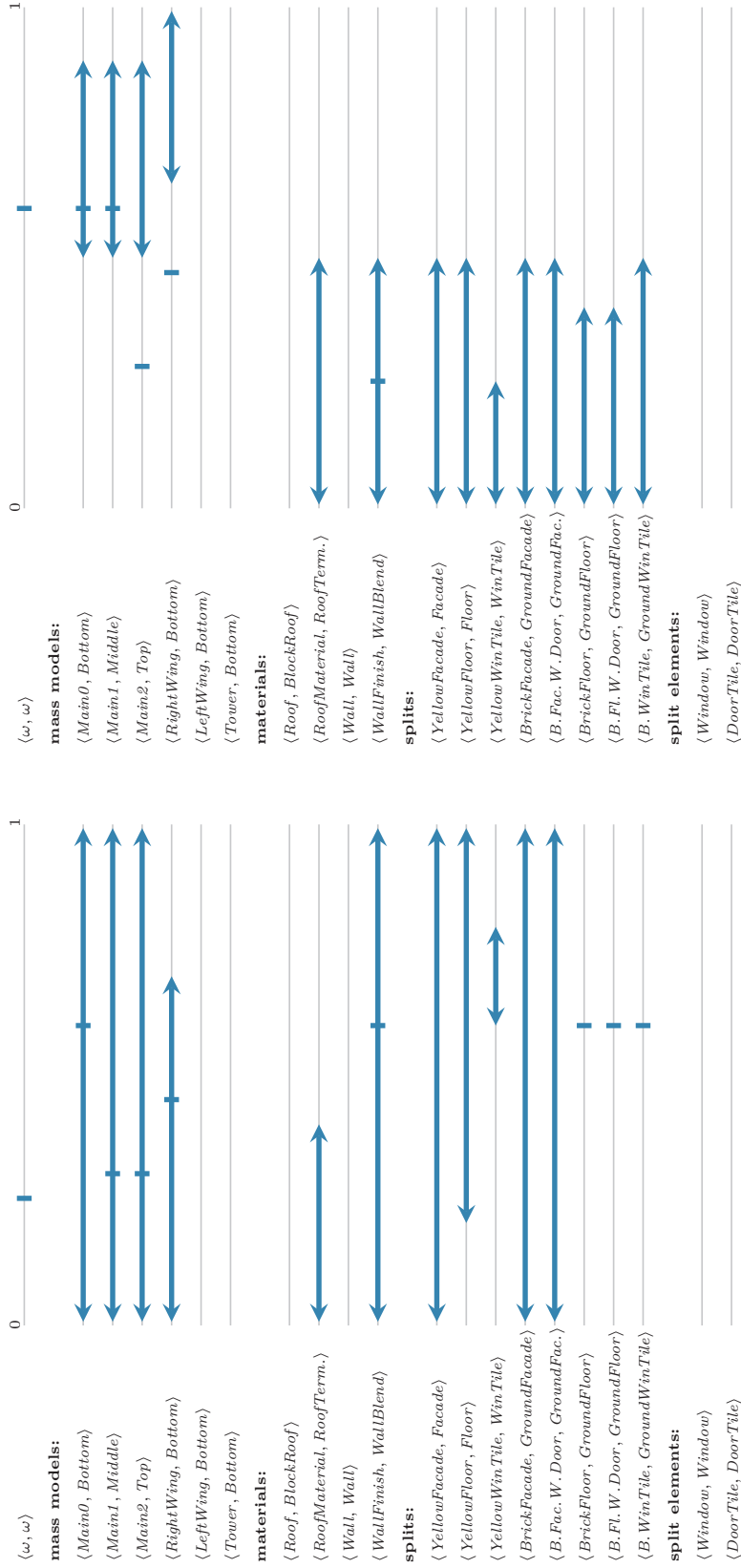


Figure A.3: Parameters for Sternwarte Chain Part 1
 Above are the settings that we use to transform the Sternwarte into the white modern building. The threshold of the axiom semantic match decides when the entire LeftWing and the Tower disappear, which happens early on. The last three semantic matches for splits perform their transformations over infinitely small time spans. This effectively results in discrete rule switches, which we represent as thresholds.

Figure A.4: Alternative Version In this version all façade and roof shapes transform before any of the mass models change. This is evident from the above list of parameters: all split and material transformations are executed over the first half of the sequence, and everything else follows in the second half. At the half-way mark, the resulting design is the Semper Sternwarte completely covered by the façade style of the white modern building. The two thresholds that are not within their $[t_s, t_e]$ range are responsible for removing the railings on top of Main2 and RightWing.

A.2 Tree L-Systems

The second example transforms between the two L-systems by Honda [Hon71] and by Aono and Kunii [AK84]. We ported the L-systems to our rule-based modeling framework. While it is the example with the lowest number of rules and with only a few semantic matches, understanding it is not trivial due to its recursive nature.

Grammar Descriptions Executing the first grammar yields a monopodial tree (Fig. A.6 top row). It has three different branching rules, A , B , and C . They each generate a straight and a lateral segment. Rule A is responsible for the tree’s vertical growth towards the sky, while rules B and C generate branches growing to the sides. The only difference between B and C is that their lateral segments lie on opposite sides of the straight segment. The second grammar grows a sympodial tree (Fig. A.6 bottom row) that is based on only two branching rules, A and B . Rule A is only applied once at the beginning for the trunk, all remaining branchings are handled by B .

Both grammars also use a rule F that instantiates a cylinder mesh to represent a segment’s geometry (F originally stood for *forward*, telling the turtle that executes the L-system to advance). All remaining rules are responsible for gradually diminishing the tree’s thickness or they implement the exit conditions for the recursive execution of the L-systems.

We only use two different tags for the symbols: `@branching` for A , B , C in the first grammar and for A , B in the second grammar, and `@segment` for F in both grammars. Putting the `@branching` tag on all branching rules means that we consider them all as interchangeable. The tags results in the six semantic matches listed in Fig. A.5.

In the following, two simplified grammars show the overall structures of both L-systems. The ellipses stand for affine transformations that are of no relevance for this explanation. Untagged in-between rules have been removed.

$$\begin{array}{ll}
 \omega \rightarrow \dots A & \omega \rightarrow \dots A \\
 A \rightarrow F \dots [\dots B] \dots A & A \rightarrow F \dots [\dots B] \dots B \\
 B \rightarrow F \dots [\dots C] \dots C & B \rightarrow F \dots [\dots B] \dots B \\
 C \rightarrow F \dots [\dots B] \dots B &
 \end{array}$$

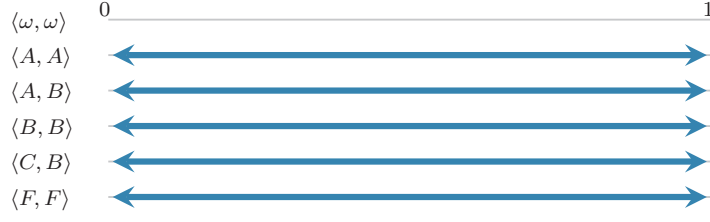


Figure A.5: Transformation Parameters for Tree L-Systems All the parameters of all semantic matches are set to their default values. Start times are set to 0 and end times to 1. No thresholds are shown since no discrete choices have to be made for this transformation: both designs do not use textures, the branch meshes are identical in both grammars, and the co-derivation steps never end up with any unmatched shapes.

Co-Derivation Steps A few iterations of the growth structure encoded by these rules are shown in the top and bottom rows of Fig. A.6. The interesting parts of the entire co-derivation happen in the co-derivation steps of our interchangeable branching rules and we explain it by means of the semantic match $\langle A, A \rangle$. The three parts of that co-derivation step are:

1. The given shape is derived with the A rules of both grammars. Both generate three successor shapes each, one F shape at the attachment point that leads to two new branching shapes.
2. The subtree collapsing step can be skipped since the simplified grammars do not generate any shapes without semantic matches.
3. For the shape matching and blending, we choose the Munkres strategy. In this case it works on two sets of three shapes each. The F shape from the first grammar can only be matched with the F shape from the second grammar. The remaining shapes are all compatible for matching. All shapes are matched and therefore all of them are blended, resulting in three shapes labeled $\langle F, F \rangle$, $\langle B, B \rangle$, and $\langle A, B \rangle$. The blending interpolates branching angles, segment lengths and radii, and color. This exact matching and blending is visualized between the first two iterations in the middle row of Fig. A.6.

Co-derivation steps for the other three semantic matches for branchings ($\langle A, B \rangle$, $\langle B, B \rangle$, and $\langle C, B \rangle$) are analogous since all branching rules always generate three similar successor shapes. Examples for these are present in columns three and four of the figure.

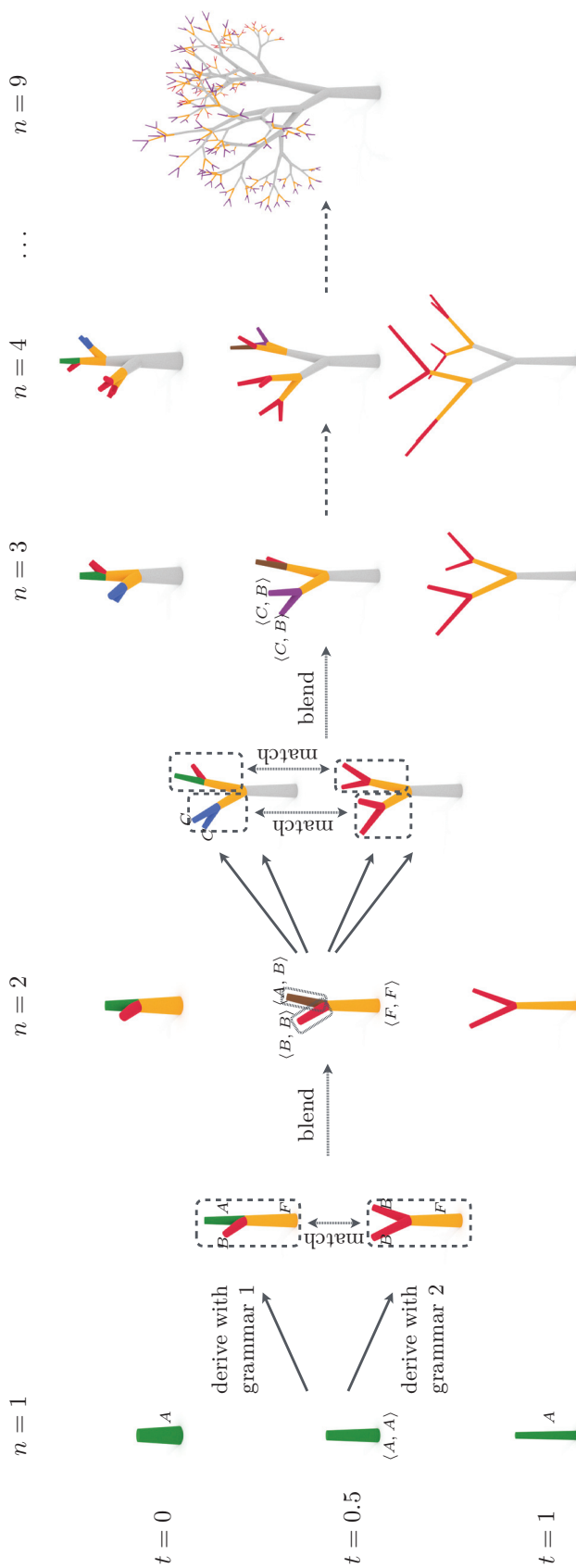


Figure A.6: Tree L-Systems Transformation. The first four co-derivation iterations of both L -systems are shown for times $t = 0$ (top), $t = 0.5$ (middle), and $t = 1.0$ (bottom). The top and bottom rows correspond to the output of the original grammars. We color code the different shapes according to their symbols: A s are green, B s are red, C s are blue, and F s are orange. Shapes generated in previous iterations are colored gray. Since blended shapes interpolate their colors we also see brown (for $\langle A, B \rangle$ blends) and purple (for $\langle C, B \rangle$ blends). Between the first and second iterations, we show how the shape matching and blending step works for the semantic match $\langle A, A \rangle$ at $t = 0.5$. The same is shown for semantic matches $\langle A, B \rangle$ and $\langle B, B \rangle$ between iterations two and three.

Appendix A. Detailed Design Transformation Result Descriptions

Parameters Note that the semantic matches $\langle B, A \rangle$ and $\langle C, A \rangle$ do not appear in the user interface (Fig. A.5) even though the tags automatically establish these correspondences. This is because such a situation can never occur. Rule A in the second grammar is only used in a co-derivation step once. That one occurrence is controlled by the $\langle A, A \rangle$ semantic match. The user interface further shows that all parameters are set to their default values for this tree L-system transformation.

Bibliography

- [AK84] AONO M., KUNII T. L.: Botanical Tree Image Generation. *IEEE Computer Graphics and Applications* (1984). [100](#), [115](#)
- [AKZM14] AVERKIOU M., KIM V., ZHENG Y., MITRA N. J.: ShapeSynth: Parameterizing Model Collections for Coupled Shape Exploration and Synthesis. *Computer Graphics Forum (Eurographics)* (2014). [56](#)
- [Ale02] ALEXA M.: Recent Advances in Mesh Morphing. *Computer Graphics Forum* (2002). [86](#)
- [ALX*14] ALHASHIM I., LI H., XU K., CAO J., MA R., ZHANG H.: Topology-Varying 3D Shape Creation via Structural Blending. *ACM Trans. Graph. (Siggraph)* (2014). [86](#), [105](#)
- [ARB07] ALIAGA D. G., ROSEN P. A., BEKINS D. R.: Style Grammars for Interactive Visualization of Architecture. *IEEE Trans. Visualization and Computer Graphics* (2007). [28](#)
- [Arn54] ARNHEIM R.: *Art and Visual Perception: A Psychology of the Creative Eye*. University of California Press, 1954. [39](#)
- [AV07] ARTHUR D., VASSILVITSKII S.: K-Means++: The Advantages of Careful Seeding. *Proc. ACM-SIAM Symp. on Discrete Algorithms* (2007). [46](#)
- [AXZ*15] ALHASHIM I., XU K., ZHUANG Y., CAO J., SIMARI P., ZHANG H.: Deformation-driven Topology-varying 3D Shape Correspondence. *ACM Trans. on Graph. (Siggraph Asia)* (2015). [86](#), [105](#)
- [BA05] BEKINS D. R., ALIAGA D. G.: Build-by-Number: Rearranging the Real World to Visualize Novel Architectural Spaces. *IEEE Visualization Conference* (2005). [28](#)

Bibliography

- [Bau72] BAUMGART B. G.: *Winged Edge Polyhedron Representation*. Tech. rep., Stanford University, 1972. [25](#)
- [BBdF10] BROCHU E., BROCHU T., DE FREITAS N.: A Bayesian Interactive Optimization Approach to Procedural Animation Design. *ACM SIGGRAPH/Eurographics Symp. Computer Animation* (2010). [56](#)
- [BBP13] BARROSO S., BESUIEVSKY G., PATOW G.: Visual Copy & Paste for Procedurally Modeled Buildings by Ruleset Rewriting. *Computers & Graphics* (2013). [26](#)
- [Bir00] BIRN J.: *Digital Lighting and Rendering*. New Riders Publishing, 2000. [39](#)
- [BIT04] BHAT P., INGRAM S., TURK G.: Geometric Texture Synthesis by Example. *Computer Graphics Forum (SGP)* (2004). [86](#)
- [BKS*05] BUSTOS B., KEIM D. A., SAUPE D., SCHRECK T., VRANIĆ D. V.: Feature-based Similarity Search in 3D Object Databases. *ACM Computing Surveys* (2005). [35](#)
- [BPC*12] BOUDON F., PRADAL C., COKELAER T., PRUSINKIEWICZ P., GODIN C.: L-Py: An L-System Simulation Framework for Modeling Plant Architecture Development Based on a Dynamic Language. *Frontiers in Plant Science* (2012). [23](#)
- [BŠMM11] BENES B., ŠT’AVA O., MĚCH R., MILLER G.: Guided Procedural Modeling. *Computer Graphics Forum (Eurographics)* (2011). [26](#)
- [BTBV96] BLANZ V., TARR M. J., BÜLTHOFF H. H., VETTER T.: What Object Attributes Determine Canonical Views? *Perception* (1996). [41](#)
- [BWKS11] BOKELOH M., WAND M., KOLTUN V., SEIDEL H.-P.: Pattern-aware Shape Deformation Using Sliding Dockers. *ACM Trans. Graph. (Siggraph)* (2011). [87](#)
- [BWS10] BOKELOH M., WAND M., SEIDEL H.-P.: A Connection Between Partial Symmetry and Inverse Procedural Modeling. *ACM Trans. Graph. (Siggraph)* (2010). [27](#), [86](#)
- [BWSK12] BOKELOH M., WAND M., SEIDEL H.-P., KOLTUN V.: An Algebraic Model for Parameterized Shape Editing. *ACM Trans. Graph. (Siggraph)* (2012). [87](#)

-
- [BYMW13] BAO F., YAN D.-M., MITRA N. J., WONKA P.: Generating and Exploring Good Building Layouts. *ACM Trans. Graph. (Siggraph)* (2013). [56](#)
- [CD01] COLLINS M., DUFFY N.: Convolution Kernels for Natural Language. *Neural Information Processing Systems* (2001). [80](#)
- [Cho56] CHOMSKY N.: Three Models for the Description of Language. *IRE Trans. Information Theory* (1956). [21](#)
- [CKGF13] CHAUDHURI S., KALOGERAKIS E., GIGUERE S., FUNKHOUSER T.: Attribit: Content Creation with Semantic Attributes. *ACM User Interface Software and Technology Symp.* (2013). [56](#)
- [CLDD09] CABRAL M., LEFEBVRE S., DACHSBACHER C., DRETTAKIS G.: Structure Preserving Reshape for Textured Architectural Scenes. *Computer Graphics Forum (Eurographics)* (2009). [87](#)
- [CSCF06] CASTELLÓ P., SBERT M., CHOVER M., FEIXAS M.: Techniques for Computing Viewpoint Entropy of a 3D Scene. *Proc. Int. Conf. on Computational Science* (2006). [36](#)
- [DBD*13] DENG B., BOUAZIZ S., DEUSS M., ZHANG J., SCHWARTZBURG Y., PAULY M.: Exploring Local Modifications for Constrained Meshes. *Computer Graphics Forum (Eurographics)* (2013). [56](#)
- [DCG10] DUTAGACI H., CHEUNG C. P., GODIL A.: A Benchmark for Best View Selection of 3D Objects. *Proc. Eurographics Workshop on 3D Object Retrieval* (2010). [35](#)
- [DLC*15] DANG M., LIENHARD S., CEYLAN D., NEUBERT B., WONKA P., PAULY M.: Interactive Design of Probability Density Functions for Shape Grammars. *ACM Trans. on Graph. (Siggraph Asia)* (2015). [18](#), [71](#), [87](#)
- [DMK*01] DENG Y., MANJUNATH B. S., KENNEY C., MOORE M. S., SHIN H.: An Efficient Color Representation for Image Retrieval. *IEEE Trans. Image Processing* (2001). [35](#)
- [EB92] EDELMAN S., BÜLTHOFF H. H.: Orientation Dependence in the Recognition of Familiar and Novel Views of Three-dimensional Objects. *Vision Research* (1992). [39](#)
- [Fed71] FEDER J.: Plex Languages. *Information Sciences* (1971). [27](#), [104](#)

Bibliography

- [GE11] GRASL T., ECONOMOU A.: GRAPE: A Parametric Shape Grammar Implementation. *Symp. on Simulation for Architecture and Urban Design* (2011). [24](#)
- [GRMS01] GOOCH B., REINHARD E., MOULDING C., SHIRLEY P.: Artistic Composition for Image Creation. *Proc. Eurographics Workshop on Rendering Techniques* (2001). [35](#), [39](#)
- [Hav05] HAVEMANN S.: *Generative Mesh Modeling*. PhD thesis, TU Braunschweig, 2005. [25](#)
- [HD03] HUANG P. W., DAI S. K.: Image Retrieval by Texture Similarity. *Pattern Recognition* (2003). [35](#)
- [HJO*01] HERTZMANN A., JACOBS C. E., OLIVER N., CURLESS B., SALESIN D. H.: Image Analogies. *Proc. of SIGGRAPH* (2001). [86](#)
- [HMG09] HAEGLER S., MÜLLER P., VAN GOOL L.: Procedural Modeling for Digital Cultural Heritage. *EURASIP Journal on Image and Video Processing* (2009). [25](#)
- [Hon71] HONDA H.: Description of the Form of Trees by the Parameters of the Tree-like Body: Effects of the Branching Angle and the Branch Length on the Shape of the Tree-like Body. *Journal of Theoretical Biology* (1971). [100](#), [115](#)
- [HWMA*10] HAEGLER S., WONKA P., MÜLLER ARISONA S., VAN GOOL L., MÜLLER P.: Grammar-based Encoding of Facades. *Computer Graphics Forum* (2010). [26](#)
- [Joh98] JOHNSON M.: PCFG Models of Linguistic Tree Representations. *Computational Linguistics* (1998). [66](#), [67](#)
- [KBK13] KRECKLAU L., BORN J., KOBELT L.: View-Dependent Realtime Rendering of Procedural Facades with High Geometric Detail. *Computer Graphics Forum (Eurographics)* (2013). [26](#)
- [KFLCO13] KLEIMAN Y., FISH N., LANIR J., COHEN-OR D.: Dynamic Maps for Exploring and Browsing Shapes. *Computer Graphics Forum (SGP)* (2013). [56](#)

-
- [KH90] KHOTANZAD A., HONG Y. H.: Invariant Image Recognition by Zernike Moments. *IEEE Trans. Pattern Analysis and Machine Intelligence* (1990). [50](#)
- [KHS10] KALOGERAKIS E., HERTZMANN A., SINGH K.: Learning 3D Mesh Segmentation and Labeling. *ACM Trans. Graph. (Siggraph)* (2010). [42](#)
- [KK11] KRECKLAU L., KOBELT L.: Procedural Modeling of Interconnected Structures. *Computer Graphics Forum (Eurographics)* (2011). [25](#)
- [KK12] KRECKLAU L., KOBELT L.: Interactive Modeling by Procedural High-Level Primitives. *Computers & Graphics* (2012). [26](#)
- [Kni94] KNIGHT T. W.: *Transformations in Design: A Formal Approach to Stylistic Change and Innovation in the Visual Arts*. Cambridge University Press, 1994. [24](#), [84](#), [86](#), [104](#)
- [KPK10] KRECKLAU L., PAVIC D., KOBELT L.: Generalized Use of Non-Terminal Symbols for Procedural Modeling. *Computer Graphics Forum* (2010). [25](#), [102](#)
- [KSI14] KOYAMA Y., SAKAMOTO D., IGARASHI T.: Crowd-powered Parameter Analysis for Visual Design Exploration. *ACM User Interface Software and Technology Symp.* (2014). [56](#)
- [KSK97] KANAI T., SUZUKI H., KIMURA F.: 3D Geometric Metamorphosis based on Harmonic Map. *Pacific Graphics* (1997). [86](#)
- [KWM15] KELLY T., WONKA P., MÜLLER P.: Interactive Dimensioning of Parametric Models. *Computer Graphics Forum (Eurographics)* (2015). [26](#)
- [Lag10] LAGA H.: Semantics-driven Approach for Automatic Selection of Best Views of 3D Shapes. *Proc. Eurographics Workshop on 3D Object Retrieval* (2010). [35](#)
- [LGL95] LERIOS A., GARFINKLE C. D., LEVOY M.: Feature-Based Volume Metamorphosis. *Proc. of SIGGRAPH* (1995). [86](#)
- [LHP11] LEBLANC L., HOULE J., POULIN P.: Component-based Modeling of Complete Buildings. *Graphics Interface 2011* (2011). [25](#)
- [Lin68] LINDENMAYER A.: Mathematical Models for Cellular Interactions in Development II. Simple and Branching Filaments with Two-Sided Inputs. *Journal of Theoretical Biology* (1968). [22](#)

Bibliography

- [Lin91] LIN J.: Divergence measures based on the Shannon entropy. *IEEE Trans. Information Theory* (1991). [70](#)
- [LLM*17] LIENHARD S., LAU C., MÜLLER P., WONKA P., PAULY M.: Design Transformations for Rule-based Procedural Modeling. *Computer Graphics Forum (Eurographics)* (2017). [19](#)
- [LOMI11] LAU M., OHGAWARA A., MITANI J., IGARASHI T.: Converting 3D Furniture Models to Fabricatable Parts and Connectors. *ACM Trans. Graph. (Siggraph)* (2011). [28](#)
- [LP96] LIU F., PICARD R. W.: Periodicity, Directionality, and Randomness: Wold features for Image Modeling and Retrieval. *IEEE Trans. Pattern Analysis and Machine Intelligence* (1996). [35](#)
- [LSK*10] LEE B., SRIVASTAVA S., KUMAR R., BRAFMAN R. I., KLEMMER S. R.: Designing with Interactive Example Galleries. *SIGCHI Conf. Human Factors in Computing Systems* (2010). [56](#)
- [LSN*14] LIENHARD S., SPECHT M., NEUBERT B., PAULY M., MÜLLER P.: Thumbnail Galleries for Procedural Models. *Computer Graphics Forum (Eurographics)* (2014). [18](#), [56](#)
- [LVJ05] LEE C. H., VARSHNEY A., JACOBS D. W.: Mesh saliency. *ACM Trans. Graph. (Siggraph)* (2005). [35](#)
- [LWW08] LIPP M., WONKA P., WIMMER M.: Interactive Visual Editing of Grammars for Procedural Architecture. *ACM Trans. Graph. (Siggraph)* (2008). [26](#)
- [LWW10] LIPP M., WONKA P., WIMMER M.: Parallel Generation of Multiple L-Systems. *Computers & Graphics* (2010). [26](#)
- [LZLM07] LIU Y., ZHANG D., LU G., MA W.-Y.: A Survey of Content-based Image Retrieval with High-level Semantics. *Pattern Recognition* (2007). [35](#)
- [MAB*97] MARKS J., ANDALMAN B., BEARDSLEY P. A., FREEMAN W., GIBSON S., HODGINS J., KANG T., MIRTICH B., PFISTER H., RUMMLER W., RYALL K., SEIMS J., SHIEBER S.: Design Galleries: A General Approach to Setting Parameters for Computer Graphics and Animation. *Proc. of SIGGRAPH* (1997). [46](#), [51](#), [54](#), [55](#)

-
- [MBG*12] MARVIE J.-E., BURON C., GAUTRON P., HIRTZLIN P., SOURIMANT G.: GPU Shape Grammars. *Pacific Graphics* (2012). [26](#)
- [MG13] MARTINOVIC A., GOOL L. V.: Bayesian Grammar Learning for Inverse Procedural Modeling. *CVPR* (2013). [29](#)
- [MGHS11] MARVIE J.-E., GAUTRON P., HIRTZLIN P., SOURIMANT G.: Render-time Procedural Per-pixel Geometry Generation. *Proc. Graphics Interface* (2011). [26](#)
- [MHS*14] MA C., HUANG H., SHEFFER A., KALOGERAKIS E., WANG R.: Analogy-Driven 3D Style Transfer. *Computer Graphics Forum (Eurographics)* (2014). [86](#)
- [MMWG11] MATHIAS M., MARTINOVIC A., WEISSENBERG J., GOOL L. V.: Procedural 3D Building Reconstruction Using Shape Grammars and Detectors. *Int. Conf. on 3D Imaging, Modeling, Processing, Visualization and Transmission* (2011). [28](#)
- [MP96] MĚCH R., PRUSINKIEWICZ P.: Visual Models of Plants Interacting with Their Environment. *Proc. of SIGGRAPH* (1996). [23](#)
- [MRF06] MALINGA B., RAICU D., FURST J.: Local vs. Global Histogram-Based Color Image Clustering. *CTI Research Symp.* (2006). [35](#)
- [MSL*11] MERRELL P., SCHKUFZA E., LI Z., AGRAWALA M., KOLTUN V.: Interactive Furniture Layout Using Interior Design Guidelines. *ACM Trans. Graph. (Siggraph)* (2011). [56](#)
- [Mun57] MUNKRES J.: Algorithms for the Assignment and Transportation Problems. *Journal of the Society for Industrial and Applied Mathematics* (1957). [92](#)
- [MWH*06] MÜLLER P., WONKA P., HAEGLER S., ULMER A., GOOL L. V.: Procedural Modeling of Buildings. *ACM Trans. Graph. (Siggraph)* (2006). [24](#), [43](#), [86](#), [87](#), [88](#), [93](#)
- [MWHG05] MÜLLER P., WONKA P., HAEGLER S., GOOL L. V.: Transformations in Architectural Design. *ACM SIGGRAPH Electronic Art and Animation Catalog* (2005). [100](#)
- [MZWG07] MÜLLER P., ZENG G., WONKA P., GOOL L. V.: Image-based Procedural Modeling of Facades. *ACM Trans. Graph. (Siggraph)* (2007). [28](#)

Bibliography

- [NGDA*16] NISHIDA G., GARCIA-DORADO I., ALIAGA D. G., BENES B., BOUSSEAU A.: Interactive Sketching of Urban Procedural Models. *ACM Trans. Graph. (Siggraph)* (2016). [28](#)
- [OLAH14] O'DONOVAN P., LİBEKS J., AGARWALA A., HERTZMANN A.: Exploratory Font Selection Using Crowdsourced Attributes. *ACM Trans. Graph. (Siggraph)* (2014). [56](#)
- [Pat12] PATOW G.: User-Friendly Graph Editing for Procedural Modeling of Buildings. *IEEE Computer Graphics and Applications* (2012). [26](#)
- [PB96] PLEMENOS D., BENAYADA M.: Intelligent Display in Scene Modeling. New Techniques to Automatically Compute Good Views. *Proc. GraphiCon* (1996). [36](#), [37](#)
- [PBS*01] PLATT J. C., BURGESS C. J. C., SWENSON S., WEARE C., ZHENG A.: Learning a Gaussian Process Prior for Automatically Generating Music Playlists. *Neural Information Processing Systems* (2001). [62](#)
- [Pel90] PELI E.: Contrast in Complex Images. *Journal of the Optical Society of America A* (1990). [38](#)
- [PHM00] PRUSINKIEWICZ P., HANAN J., MĚCH R.: An L-System-Based Plant Modeling Language. *Proc. Int. Workshop on Applications of Graph Transformations with Industrial Relevance* (2000). [23](#)
- [PJM94] PRUSINKIEWICZ P., JAMES M., MĚCH R.: Synthetic Topiary. *Proc. of SIGGRAPH* (1994). [23](#)
- [PKMH00] PRUSINKIEWICZ P., KARWOWSKI R., MĚCH R., HANAN J.: L-Studio/CPFG: A Software System for Modeling Plants. *Proc. Int. Workshop on Applications of Graph Transformations with Industrial Relevance* (2000). [23](#)
- [PL90] PRUSINKIEWICZ P., LINDENMAYER A.: *The Algorithmic Beauty of Plants*. Springer, 1990. [22](#), [102](#)
- [PM01] PARISH Y. I. H., MÜLLER P.: Procedural Modeling of Cities. *Proc. of SIGGRAPH* (2001). [23](#)
- [PMKL01] PRUSINKIEWICZ P., MÜNDERMANN L., KARWOWSKI R., LANE B.: The Use of Positional Information in the Modeling of Plants. *Computer Graphics* (2001). [23](#)

-
- [PR69] PFALTZ J. L., ROSENFELD A.: Web Grammars. *Proc. Int. Joint Conf. on Artificial Intelligence* (1969). [27](#)
- [Pru86] PRUSINKIEWICZ P.: Graphical Applications of L-systems. *Proc. Graphics Interface/Vision Interface* (1986). [22](#), [86](#), [87](#)
- [PSG*06] PODOLAK J., SHILANE P., GOLOVINSKIY A., RUSINKIEWICZ S., FUNKHOUSER T.: A Planar-Reflective Symmetry Transform for 3D Shapes. *ACM Trans. Graph. (Siggraph)* (2006). [35](#)
- [RKT*12] RIEMENSCHNEIDER H., KRISPEL U., THALLER W., DONOSER M., HAVE-MANN S., FELLNER D. W., BISCHOF H.: Irregular Lattices for Complex Shape Grammar Facade Parsing. *CVPR* (2012). [28](#)
- [RMGH15] RITCHIE D., MILDENHALL B., GOODMAN N. D., HANRAHAN P.: Controlling Procedural Modeling Programs with Stochastically-Ordered Sequential Monte Carlo. *ACM Trans. Graph. (Siggraph)* (2015). [28](#)
- [RP12] RIU A., PATOW G.: Bringing Direct Local Control to Interactive Visual Editing of Procedural Buildings. *Spanish Computer Graphics Conference* (2012). [26](#)
- [RW06] RASMUSSEN C. E., WILLIAMS C. K. I.: *Gaussian Processes for Machine Learning*. MIT Press, 2006. [62](#)
- [ŠBM*10] ŠT'AVA O., BENES B., MĚCH R., ALIAGA D. G., KRIŠTOF P.: Inverse Procedural Modeling by Automatic Generation of L-systems. *Computer Graphics Forum (Eurographics)* (2010). [27](#), [86](#)
- [SdGP*15] SOLOMON J., DE GOES F., PEYRÉ G., CUTURI M., BUTSCHER A., NGUYEN A., DURAND T., GUIBAS L.: Convolutional Wasserstein Distances: Efficient Optimal Transportation on Geometric Domains. *ACM Trans. Graph. (Siggraph)* (2015). [105](#)
- [Sed86] SEDERBERG T. W.: Free-Form Deformation of Solid Geometric Models. *Computer Graphics* (1986). [25](#)
- [SG71] STINY G. N., GIPS J.: Shape Grammars and the Generative Specification of Painting and Sculpture. *IFIP Congress* (1971). [23](#)
- [SKK*12] STEINBERGER M., KAINZ B., KERBL B., HAUSWIESNER S., KENZEL M., SCHMALSTIEG D.: Softshell: Dynamic Scheduling on GPUs. *ACM Trans. Graph. (Siggraph)* (2012). [26](#)

Bibliography

- [SKK*14a] STEINBERGER M., KENZEL M., KAINZ B., MÜLLER J., PETER W., SCHMALSTIEG D.: Parallel Generation of Architecture on the GPU. *Computer Graphics Forum (Eurographics)* (2014). [26](#)
- [SKK*14b] STEINBERGER M., KENZEL M., KAINZ B., WONKA W., SCHMALSTIEG D.: On-the-Fly Generation and Rendering of Infinite Cities on the GPU. *Computer Graphics Forum (Eurographics)* (2014). [26](#)
- [SLF*11] SECORD A., LU J., FINKELSTEIN A., SINGH M., NEALEN A.: Perceptual models of viewpoint preference. *ACM Trans. Graph. (Siggraph)* (2011). [35](#), [42](#), [47](#), [48](#)
- [SM78] STINY G. N., MITCHELL W. J.: The Palladian Grammar. *Environment and Planning B: Planning and Design* (1978). [23](#)
- [SM15] SCHWARZ M., MÜLLER P.: Advanced Procedural Modeling of Architecture. *ACM Trans. Graph. (Siggraph)* (2015). [25](#)
- [SMBC13] SILVA P. B., MÜLLER P., BIDARRA R., COELHO A.: Node-Based Shape Grammar Representation and Editing. *Proc. PCG* (2013). [26](#)
- [Smi97] SMITH J. T.: *Remarks on Rural Scenery*. Nathaniel Smith et al., 1797. [38](#)
- [SMR04] SEMPER G., MALLGRAVE H., ROBINSON M.: *Style: Style in the Technical and Tectonic Arts; or, Practical Aesthetics*. Getty Research Institute, 2004. [84](#)
- [SO94] STOLCKE A., OMOHUNDRO S.: Inducing Probabilistic Grammars by Bayesian Model Merging. *Int. Conf. Grammatical Inference* (1994). [28](#)
- [SP16] SANTONI C., PELLACINI F.: gTangle: A Grammar for the Procedural Generation of Tangle Patterns. *ACM Trans. on Graph. (Siggraph Asia)* (2016). [25](#)
- [ŠPK*14] ŠT’AVA O., PIRK S., KRATT J., CHEN B., MĚCH R., DEUSSEN O., BENES B.: Inverse Procedural Modelling of Trees. *Computer Graphics Forum* (2014). [28](#)
- [SSCO09] SHAPIRA L., SHAMIR A., COHEN-OR D.: Image Appearance Exploration by Model-Based Navigation. *Computer Graphics Forum (Eurographics)* (2009). [56](#)
- [ST90] SAITO T., TAKAHASHI T.: Comprehensible Rendering of 3-D Shapes. *Computer Graphics* (1990). [36](#)

-
- [Sti75] STINY G. N.: *Pictorial and Formal Aspects of Shape and Shape Grammars and Aesthetic Systems*. PhD thesis, University of California, Los Angeles, 1975. [23](#)
- [Sti80] STINY G. N.: Introduction to shape and shape grammars. *Environment and Planning B* (1980). [23](#)
- [Sti82] STINY G. N.: Spatial Relations and Grammars. *Environment and Planning B* (1982). [24](#), [86](#)
- [STKP11] SIMON L., TEBOUL O., KOUTSOURAKIS P., PARAGIOS N.: Random Exploration of the Procedural Space for Single-View 3D Modeling of Buildings. *IJCV* (2011). [28](#), [60](#)
- [Sto94] STOLCKE A.: *Bayesian Learning of Probabilistic Language Models*. PhD thesis, University of California, Berkeley, 1994. [28](#), [66](#)
- [TGY*09] TALTON J. O., GIBSON D., YANG L., HANRAHAN P., KOLTUN V.: Exploratory Modeling with Collaborative Design Spaces. *ACM Trans. on Graph. (Siggraph Asia)* (2009). [54](#), [55](#), [56](#), [60](#), [67](#), [72](#), [75](#), [76](#), [87](#)
- [TKS*11] TEBOUL O., KOKKINOS I., SIMON L., KOUTSOURAKIS P., PARAGIOS N.: Shape Grammar Parsing via Reinforcement Learning. *CVPR* (2011). [28](#)
- [TKZ*13] THALLER W., KRISPEL U., ZMUGG R., HAVEMANN S., FELLNER D. W.: Shape Grammars on Convex Polyhedra. *Computers & Graphics* (2013). [25](#)
- [TLL*11] TALTON J. O., LOU Y., LESSER S., DUKE J., MĚCH R., KOLTUN V.: Metropolis Procedural Modeling. *ACM Trans. Graph. (Siggraph)* (2011). [28](#)
- [TSK*10] TEBOUL O., SIMON L., KOUTSOURAKIS P., , PARAGIOS N.: Segmentation of Building Facades using Procedural Shape Prior. *CVPR* (2010). [28](#)
- [TYK*12] TALTON J. O., YANG L., KUMAR R., LIM M., GOODMAN N. D., MĚCH R.: Learning Design Patterns with Bayesian Grammar Induction. *ACM User Interface Software and Technology Symp.* (2012). [28](#), [84](#), [86](#), [104](#)
- [UIM12] UMETANI N., IGARASHI T., MITRA N. J.: Guided Exploration of Physically Valid Shapes for Furniture Design. *ACM Trans. Graph. (Siggraph)* (2012). [56](#)

Bibliography

- [Váz03] VÁZQUEZ P.-P.: *On the Selection of Good Views and its Application to Computer Graphics*. PhD thesis, Technical University of Catalonia, 2003. [40](#)
- [VFSH01] VÁZQUEZ P.-P., FEIXAS M., SBERT M., HEIDRICH W.: Viewpoint Selection using Viewpoint Entropy. *Proc. Vision Modeling and Visualization Conf.* (2001). [34](#), [40](#)
- [VFSL02] VÁZQUEZ P.-P., FEIXAS M., SBERT M., LLOBET A.: Viewpoint Entropy: a New Tool for Obtaining Good Views of Molecules. *Proc. Symp. on Data Visualisation* (2002). [35](#), [41](#)
- [WOD09] WHITING E., OCHSENDORF J., DURAND F.: Procedural Modeling of Structurally-Sound Masonry Buildings. *ACM Trans. on Graph. (Siggraph Asia)* (2009). [25](#)
- [WP95] WEBER J., PENN J.: Creation and Rendering of Realistic Trees. *Proc. of SIGGRAPH* (1995). [70](#)
- [WRPG13] WEISSENBERG J., RIEMENSCHNEIDER H., PRASAD M., GOOL L. V.: Is there a Procedural Logic to Architecture? *CVPR* (2013). [29](#)
- [WWSR03] WONKA P., WIMMER M., SILLION F., RIBARSKY W.: Instant Architecture. *ACM Trans. Graph. (Siggraph)* (2003). [24](#), [93](#)
- [WYD*14] WU F., YAN D.-M., DONG W., ZHANG X., WONKA P.: Inverse Procedural Modeling of Facade Layouts. *ACM Trans. Graph. (Siggraph)* (2014). [28](#)
- [XZCOC12] XU K., ZHANG H., COHEN-OR D., CHEN B.: Fit and Diverse: Set Evolution for Inspiring 3D Shape Galleries. *ACM Trans. Graph. (Siggraph)* (2012). [56](#)
- [YKG*09] YUE K., KRISHNAMURTI R., GOBLER F., YUE K., KRISHNAMURTI R., GROBLER F.: Computation-Friendly Shape Grammars. *Proc. CAAD Futures* (2009). [24](#)
- [YYPM11] YANG Y.-L., YANG Y.-J., POTTMANN H., MITRA N. J.: Shape Space Exploration of Constrained Meshes. *ACM Trans. on Graph. (Siggraph Asia)* (2011). [56](#)
- [ZR72] ZAHN C. T., ROSKIES R. Z.: Fourier descriptors for plane closed curves. *IEEE Trans. Computers* (1972). [49](#)

- [ZTK*13] ZMUGG R., THALLER W., KRISPEL U., EDELSBRUNNER J., HAVEMANN S., FELLNER D. W.: Deformation-Aware Split Grammars for Architectural Models. *International Conference on Cyberworlds* (2013). [25](#)
- [ZTK*14] ZMUGG R., THALLER W., KRISPEL U., EDELSBRUNNER J., HAVEMANN S., FELLNER D. W.: Procedural Architecture Using Deformation-Aware Split Grammars. *The Visual Computer* (2014). [25](#)

Stefan Lienhard – Minimal CV

Education

- 2011-2017 **PhD in Computer Graphics (Advisor: Prof. Mark Pauly)**
Ecole Polytechnique Fédérale de Lausanne (EPFL)
- 2008-2010 **MSc in Information Technology & Electrical Engineering**
Eidgenössische Technische Hochschule Zürich (ETH)
- 2004-2007 **BSc in Information Technology & Electrical Engineering**
Eidgenössische Technische Hochschule Zürich (ETH)

Publications

- 2017 **Design Transformations for Rule-based Procedural Modeling**
Stefan Lienhard, Cheryl Lau, Pascal Müller, Peter Wonka, Mark Pauly
Computer Graphics Forum (Eurographics)
- 2015 **Interactive Design of Probability Density Functions for Shape Grammars**
Minh Dang, Stefan Lienhard, Duygu Ceylan, Boris Neubert, Peter Wonka, and Mark Pauly
ACM Transactions on Graphics (Siggraph Asia)
- 2014 **Thumbnail Galleries for Procedural Models**
Stefan Lienhard, Matthias Specht, Boris Neubert, Mark Pauly, and Pascal Müller
Computer Graphics Forum (Eurographics)
- 2011 **A Full Bi-Tensor Neural Tractography Algorithm Using the Unscented Kalman Filter**
Stefan Lienhard, James G. Malcolm, Carl-Frederik Westin, and Yogesh Rathi
EURASIP Journal on Advances in Signal Processing